

Real-time Processor Architectures for Worst Case Execution Time Reduction

(Submitted for the degree of Doctor of Philosophy)

Jack Whitham

*Department of Computer Science,
University of York*

January 2008

Abstract

In a real-time system, programs must respond to external events in a timely fashion, completing all required subtasks before a deadline elapses. Worst case execution time (WCET) analysis is used to assure that deadlines are always met by determining the maximum period of time required to execute parts of a program. WCET analysis has been extensively studied, but existing techniques do not work well with CPUs that make use of dynamic execution optimisation features such as caches and superscalar out-of-order issue units. These features work well for average case execution time (ACET) reduction, but not for WCET reduction, since software must model the worst possible behaviour of the dynamic features to find the WCET.

This thesis explores the architectural issues relating to these problems, then characterises a class of CPU architecture that is explicitly designed for WCET reduction. An implementation of this architecture enables WCET reduction by allowing programs to move worst-case execution paths into a microprogram store, which acts as a scratchpad for parallel microinstructions. The exploitation of instruction-level parallelism (ILP) across basic block boundaries is enabled by the use of a superblock scheduler, which allows the architecture to reduce WCET further than previous CPU architectures intended to facilitate timing analysis.

Instances of the new architecture class are implemented and tested using field-programmable gate array (FPGA) hardware. A WCET reduction algorithm is proposed, implemented and tested based on the implicit path enumeration technique (IPET) for WCET analysis. Experiments are used to compare the algorithm and architecture with previous work.

Table of Contents

1	Introduction	1
1.1	Thesis Aims	3
1.2	Digital Appendix	4
1.3	Thesis Structure	4
2	Literature	5
2.1	About Timing Analysis	5
2.1.1	WCET Estimation	6
2.1.2	Reducing Pessimism in WCET analysis	8
2.1.3	The Translation Step	11
2.1.4	CPU Modelling	11
2.2	A Brief History of CPU Technology	12
2.2.1	The Memory Bottleneck	12
2.2.2	The Instruction Rate Bottleneck	15
2.2.3	Effects of Dynamic Features	20
2.3	WCET Analysis of a Complex CPU	20
2.3.1	Modelling a Pipeline in Isolation	21
2.3.2	Modelling an Instruction Cache in Isolation	21
2.3.3	Modelling Pipelines and Instruction Caches Together	24
2.3.4	Modelling Multiple-Issue Pipelines	25
2.3.5	Modelling Branch Prediction	25
2.3.6	Practical Issues for CPU Modelling	26
2.3.7	Timing Anomalies	28
2.3.8	Accounting for Timing Anomalies	30
2.3.9	Trends Leading Away From Complex Models	30
2.3.10	Alternative Implementation Platforms	32
2.3.11	Statistical Analysis Methods	34
2.3.12	Multitasking	35
2.3.13	Summary	36
2.4	Improving Worst-Case Performance Using Software	37
2.4.1	Manual Optimisation	38
2.4.2	Automatic Optimisation	40
2.4.3	Manual Run-time Specialisation	43
2.4.4	Summary	44
2.5	Application-Specific Hardware	44
2.5.1	Hardware Description Languages	45

2.5.2	ASICs and FPGAs	45
2.5.3	Summary	49
2.6	Improving Worst-Case Performance Using Hardware	49
2.6.1	Co-processors	49
2.6.2	Application-Specific Instruction Processors (ASIPs)	50
2.6.3	ASIP Techniques	51
2.6.4	Classical Co-design	52
2.6.5	Using FPGA-based Run-time Reconfiguration	53
2.6.6	Coarse-Grained Reconfigurable Architectures	56
2.6.7	CGRAs within a CPU Core	62
2.6.8	Compiling Code for an RFU	65
2.7	Summary	71
3	Motivation and Hypothesis	73
3.1	Revised Thesis Aims	74
3.2	Statement Of Requirements	76
3.3	Approach	76
3.4	Hypothesis	77
3.5	Evaluation Criteria	77
4	Architecture	79
4.1	Introducing TARGET and the WCET Reduction Process	79
4.2	Construction of TARGET Architecture	80
4.2.1	Starting Point - CPU	81
4.2.2	A Simple CPU, plus an RFU	83
4.2.3	Communications Infrastructure	85
4.2.4	Controlling CPU Operations	87
4.2.5	TARGET Architecture Characterisation	87
4.3	Theoretical Evaluation	90
4.3.1	Full Support For General Software Programs	90
4.3.2	Basic Block Timing Invariance	90
4.3.3	Efficient Optimisation Process	90
4.3.4	Scalability	90
4.3.5	WCET Reduction versus Previous Work	91
4.3.6	TARGET Bottleneck	92
4.3.7	Summary	93
5	Static Implementation	95
5.1	Simulation	97
5.1.1	Existing Simulators	97
5.1.2	Simulator Validation	98
5.1.3	Simulating an RFU	99
5.2	MCGREP-1 TARGET Generator	99
5.2.1	Parameterisable Features	99
5.2.2	Non-Parameterisable Features	102
5.2.3	Microarchitecture	105

5.2.4	Microcode Generator	111
5.2.5	Simulator	114
5.3	MCGREP-1 Evaluation	114
5.3.1	Evaluation for Correctness	116
5.3.2	Evaluation against the TARGET parameters	118
5.3.3	Evaluation against the Requirements	119
5.3.4	Invariant basic block execution times - calculations	119
5.3.5	Invariant basic block execution times - experiment	121
5.3.6	WCET Reduction versus Previous Work	126
5.3.7	Other Requirements	129
5.3.8	Summary	131
6	Extensible Implementation	133
6.1	Automatic Microcode Generation Methods for TARGET	133
6.1.1	Generating VLIW Code	135
6.1.2	Acyclic Scheduling	136
6.1.3	Cyclic Scheduling	144
6.1.4	Using VLIW Scheduling Algorithms for TARGET	145
6.2	MCGREP-2 TARGET Generator	145
6.2.1	Features	146
6.2.2	Microarchitecture - Top Level	150
6.2.3	Microarchitecture - Unit Level	154
6.2.4	Microarchitecture - Programming Interface	156
6.2.5	Simulator	158
6.3	MCGREP-2 Trace-style Scheduler	160
6.3.1	Trace Formation	161
6.3.2	Operation List Building	163
6.3.3	Compaction	164
6.3.4	Bookkeeping and Exit Cleanup	168
6.3.5	Output Stage	168
6.4	MCGREP-2 Evaluation	170
6.4.1	Correctness Evaluation: Machine Code Operation	170
6.4.2	Correctness Evaluation: Microcode Generator	172
6.4.3	Correctness Evaluation: Machine Code and Microcode	174
6.4.4	Correctness Evaluation: Microcode in Hardware	178
6.4.5	Evaluation against the TARGET parameters	178
6.4.6	Evaluation against the Requirements	180
6.4.7	Invariant basic block execution times	180
6.4.8	WCET Reduction versus Previous Work	181
6.4.9	Efficient optimisation process	184
6.4.10	Scalability - programs of any size	185
6.4.11	Scalability - increasing resources	188
6.4.12	Usage of FPGA Space	190
6.4.13	Optimisation Possibilities	192
6.4.14	Summary	195

7	WCET Reduction using IPET	197
7.1	WCET Reduction	197
7.1.1	Identifying Worst-Case Paths	201
7.1.2	Modelling the Effects of a WCET Reduction	202
7.1.3	Modelling the Functionality of a Trace	202
7.1.4	Modelling Worst-Case Flow of Traces	203
7.1.5	Multiple Traces and the WCET	205
7.1.6	Constraints	205
7.1.7	Equivalence to the Puschner and Schedl model	205
7.1.8	Adapting the Puschner and Schedl proof	206
7.1.9	Allocating the Microprogram Store Space	206
7.1.10	Microprogram Store Space Allocation Algorithm	208
7.1.11	Phase 1: Selecting Good Microprogram Starting Points and Evaluating Microprograms	208
7.1.12	Phase 2: Selecting Microprograms	212
7.1.13	Summary	213
7.2	Implementing and Testing the WCET Reduction Approach	213
7.2.1	Obtaining the T-graph	213
7.2.2	Custom Microcode-Aware WCET Analysis using IPET	214
7.2.3	Solving the Integer Linear Program	216
7.2.4	Finding Candidates	217
7.2.5	Microprogram Selection	217
7.2.6	Testing the Implementation	218
7.2.7	Global tests	219
7.2.8	Local tests	223
7.2.9	Explicit Path Enumeration Tests	224
7.2.10	Summary	227
7.3	Effectiveness of the WCET Reduction Algorithm	227
7.3.1	Maximum Path Length L , and Microprogram Store Space C_{max}	228
7.3.2	Maximum Number of Search Iterations, H	228
7.3.3	Number of Trace Start Points to Test, W	230
7.3.4	Assumptions	232
7.3.5	Improvements: Search Strategy	234
7.3.6	Improvements: Score Heuristic	235
7.4	Implementing Scratchpad Allocation	237
7.4.1	Implementing Suhendra's Instruction Scratchpad Allocation Algorithm	238
7.5	Summary	239
8	Evaluation	241
8.1	Exploiting ILP Within a Single Basic Block	242
8.2	Comparison with Instruction Scratchpads	243
8.3	Comparison Experiment	243
8.4	Additional Experiments	248
8.5	Summary	249

9	Conclusion	251
9.1	Summary of Findings	251
9.2	Future Work	253
9.3	Conclusion	255
10	References	257
A	Digital Appendix Documentation	277
A.1	Overview	277
A.2	Required/recommended Software Environment	279
A.3	Installing the Archive Software	280
A.4	Building the MCGREP-1 Test Cases	281
A.5	Using the MCGREP-1 Hardware Generator	283
A.6	Using the MCGREP-1 Simulator	283
A.7	Using the MCGREP-2 Test Cases	284
A.8	Using the MCGREP-2 Hardware Generator	284
A.9	Using the MCGREP-2 Simulator	287
A.10	Extending MCGREP-2	287
A.11	Connecting an FPGA	288
A.12	Appendix Software Tests	289
A.13	Third Party Software and Hardware	289
B	Experimental Data and Test Examples	291
C	OpenRISC ORBIS32 Quick Reference	295
D	Microblaze Extensions for MCGREP-2	299
D.1	ALU Features	300
D.2	ISA and Built-in Microprogram	301
D.3	Interrupt and Status Register Support	301
D.4	Trace-style Scheduler	302
D.5	Repeating Experiments and Tests	302
D.6	Booting uClinux	303
E	Implicit Path Enumeration Technique - Example	307
F	WCET Reduction Algorithm - Example	311
G	Partitioning Support for the WCET Reduction Algorithm	313
G.1	Adding Partitioning Support For Microprograms	313
G.2	An Automatic Partitioning Strategy	314
G.3	Comparison with Partitioned Scratchpad Configurations	316

List of Tables

2.1	Examples of IDL statements	9
2.2	Puschner's integer linear programming model	10
2.3	Example CISC-style instructions	15
2.4	Li's cache-aware integer linear programming model	23
2.5	Healy's hazard model	25
2.6	FPGA and MPGA comparison	48
2.7	The scale of classical co-design systems.	53
2.8	Examples of penalties incurred by crossing level boundaries.	69
4.1	The parameters of the architecture.	88
5.1	Properties that are hard to evaluate without a model	95
5.2	SimpleScalar tools	98
5.3	Functional groups in the OpenRISC ORBIS32 ISA	103
5.4	Special microprograms for CPUs generated by MCGREP-1.	104
5.5	Parameters for initial generator implementation.	105
5.6	Types of sign extension required by OpenRISC ORBIS32 ISA.	107
5.7	Types of ALU operation required by OpenRISC ORBIS32 ISA.	107
5.8	uPC branch commands	111
5.9	Benchmarks	117
5.10	Instruction timings	120
5.11	Least-squares derivation of unknowns	125
5.12	Number of instructions executed for each benchmark.	127
5.13	IPC for each CPU	128
5.14	Core size and speed comparison	130
6.1	Exceptions that may be raised	143
6.2	Parameters for MCGREP-2 generator implementation	146
6.3	Special microprograms for CPUs generated by MCGREP-2.	149
6.4	uPC branch commands	152
6.5	Types of hook provided by the MCGREP-2 simulator.	161
6.6	Benchmarks	172
6.7	Correctness Evaluation	177
6.8	Instruction timings	180
6.9	New measurement opcodes	184
6.10	Trace build timings	185
6.11	Core size and speed comparison	191

6.12	No System Cycle	193
7.1	WCET reduction parameters	218
7.2	Test programs for WCET reduction.	219
7.3	Global test results 1	221
7.4	Global test results 2	222
7.5	“EPET” Results, 2	226
7.6	Evaluating assumptions	233
7.7	WCET reductions with different trace lengths	234
8.1	Architectural Configuration	244
8.2	Parameters used for the experiment described in section 8.3.	245
8.3	Detailed results for section 8.3	246
A.1	Configuration Parameters	286
B.1	Results of interference experiment	292
B.2	Benchmark execution times	292
B.3	Benchmark execution times	293
B.4	Multiple context timings	293
B.5	Improvements with different architectures	294
D.1	Microblaze addition and subtraction instructions	300
D.2	Irregular instructions	301
D.3	Petalinux memory map	305
E.1	Execution Time Costs	308
E.2	Conservation of Flow Constraints	310
E.3	Worst-case Flow Values	310
F.1	Possible trace starting points	312
F.2	Execution Costs	312
F.3	WCET Reductions from BB5	312
G.1	Optimal region count for environment 1	318
G.2	Optimal region count for environment 2	319

List of Figures

2.1	WCET Problem Domains	7
2.2	Sample execution path graph	7
2.3	Infeasible path example	8
2.4	Example: annotations used to bound WCET analysis	9
2.5	CPU core frequencies: 1980-2005	13
2.6	Example of ILP	16
2.7	Arnold's WCET approach	22
2.8	Example of conservative behaviour in Arnold's approach	23
2.9	Suhendra's greedy heuristic	33
2.10	Puaut and Pais algorithm	33
2.11	Specialised instruction example	39
2.12	A simple hotspot	40
2.13	A sparse matrix multiplication.	43
2.14	An island-style FPGA	47
2.15	A shift register, used to configure some FPGA components	47
2.16	Example of ASIP code	51
2.17	The idealised co-design process.	52
2.18	Molen source code annotations	54
2.19	Pictures of Karlsruhe project architecture	54
2.20	Simplified CGRA	56
2.21	KressArray example	58
2.22	Binary data, embedded in C source	59
2.23	The Rapid CGRA interconnect	59
2.24	Diagram of Piperench CGRA	60
2.25	The Garp architecture	61
2.26	Idealised RFU architecture	62
2.27	The Chimaera Architecture	63
2.28	Two-level indirection in QM-1	66
2.29	An architecture represented as a multi-level structure	69
4.1	WCET Reduction Process	80
4.2	Simple CPU data path	82
4.3	Contrasting array shapes	84
4.4	Abstract high-level view of architecture	86
4.5	High-level view of architecture with microprogram stores	88
4.6	WCET Reduction Process plus CPU and RFU	89

4.7	Transformation applied to execution path graph	91
5.1	WCET Reduction Process: CPU/RFU Implementation	96
5.2	Components required	97
5.3	Tradeoffs in simulator design	98
5.4	Verification of a simulator	99
5.5	Sample usage of microcode container	101
5.6	Part of the OpenRISC ISA implementation	104
5.7	Architecture diagram	106
5.8	Address mux: detail	108
5.9	Decoding tree for ORBIS32	110
5.10	Branch to microprogram instruction	111
5.11	Diagram of microcode generator	112
5.12	GUI for micro-operation scheduling	113
5.13	WCET Reduction Process with MCGREP -1	115
5.14	Test diagram	117
5.15	Debugging monitor architecture	118
5.16	Integrated debugging monitor	118
5.17	Basic blocks in the sample WC path fragment (machine code)	120
5.18	Microprogram sequence for WC path fragment	121
5.19	Basic blocks in the sample WC path fragment (microprogram)	122
5.20	Inter-task Interference Scenario	122
5.21	Results of interference experiment	124
5.22	Benchmark execution times	128
6.1	WCET Reduction Process: Extensible CPU/RFU Implementation	134
6.2	VLIW scheduler overview	136
6.3	Trace Scheduling Example	137
6.4	Data Flow Graph	138
6.5	Compaction Process	139
6.6	Tail Duplication Example	141
6.7	The role of the architecture interface	147
6.8	Microexecution Trace	148
6.9	System Microprogram	149
6.10	Top level diagram	151
6.11	Architecture diagram	155
6.12	Microprogramming Example	157
6.13	Simulator decoding steps	158
6.14	Architecture of MCGREP-2 simulator.	160
6.15	Structure of a trace	164
6.16	WCET Reduction Process: MCGREP-2 Implementation	171
6.17	Checkpoint and hotspot marker instructions	174
6.18	Marker added to C code	176
6.19	Benchmark execution times	183
6.20	Multiple contexts example	186

6.21	Multiple context timings	187
6.22	Effects of changing array size	189
6.23	Effects of increasing trace length	189
6.24	Implementations of TARGET	195
7.1	WCET Reduction Process: Missing Components	198
7.2	A trace as part of a T-graph	203
7.3	Score heuristic	210
7.4	WC path finding heuristic	210
7.5	The <i>Find.Candidates</i> procedure finds and evaluates candidate traces.	211
7.6	Framework for E.T. measurements	220
7.7	Bubble sort C source.	225
7.8	Developer-specified constraints for Figure 7.7.	225
7.9	“EPET” Results, 1	226
7.10	Length and Microprogram Store Space	229
7.11	Effects of changing W	231
7.12	Basic block graph of <code>bubble</code> (Figure 7.7).	231
7.13	Effects of changing W : improved heuristic	237
7.14	Hybrid Architecture	238
7.15	WCET Reduction Process: Scratchpads	240
8.1	Results for section 8.3	245
8.2	Final WCET Reduction Process	247
8.3	Improvements with different architectures	248
A.1	Software Map	278
A.2	MM1 Photograph	285
B.1	Hotspot Example	291
B.2	Bubble sort inner loop, as ORBIS32 assembly. C source appears in Figure 7.7.	293
D.1	Benchmark execution times	304
E.1	Sample T-graph	307
E.2	GLPK Commands for Example	309
F.1	Example Trace	312
G.1	Extended <i>Find.Candidates</i>	315
G.2	Percentage WCET reduction for each case	318

For Jillian

Acknowledgements

This work would not have been possible without the advice of my supervisor, Dr Neil Audsley, who has read many drafts of this work and earlier publications. I would also like to thank current and former staff and students of the Real-Time Systems Group at the University of York for their support. Special thanks to Adam Betts for his advice regarding WCET analysis issues, to Andrew Borg, Rui Gao, Ian Gray, Nick Lay and Ameet Patil for their help in the lab, and to Michael Ward and Andy Wellings for their advice on publications. Thanks also go to my examiners, Iain Bate and Mark Zwolinski, who made many suggestions for improvements to this work.

I must also thank the authors of all the work cited in chapter 10, and the authors of the freely redistributable software and IP used by my experiments (appendix A), including Python [208], gcc [94], and the OpenRISC CPU core [150]. This work would have been significantly more difficult without access to free software and hardware designs.

Finally, I would like to thank my family and friends for putting up with me as I completed this work. In particular I would like to thank my fiancée Jillian, my parents Susan and Peter, and my friends from UoY.

Declaration

This thesis has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree other than Doctor of Philosophy of the University of York. This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by explicit references.

Some parts of the thesis are also described by existing publications. The MCGREP-1 implementation and experiments (chapter 5) are summarised by [276]. The MCGREP-2 simulator (section 6.2.5) is described by [277]. The IPET-based WCET reduction algorithm and experiments (chapter 7) and the evaluation against instruction scratchpads (chapter 8) are described by [278]. All of these publications were written by myself with advice from my supervisor, Dr Neil Audsley.

I hereby give consent for my thesis to be made available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Chapter 1

Introduction

Almost every modern home and workplace contains many computers. Some of these computers are immediately recognisable as laptops and desktop PCs, but many are hidden within *intelligent devices* that use software to implement functionality. These computers are *embedded systems*. They have been widely used in place of discrete electronics within televisions, radios, video cassette recorders and washing machines for more than two decades. A common definition of the term is as follows ([25], “Embedded System”):

“A combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function. In some cases, embedded systems are part of a larger system or product, as in the case of an anti-lock braking system in a car.”

Intuitively, including a computer within some devices might seem like overkill. Does a toaster *really* need a microchip? Clearly not - toasters predate microelectronics, as do televisions, radios and video recorders. But the embedded system is an answer to a problem: *how can the manufacturing cost be minimised?*

Perhaps surprisingly, using an embedded system actually *reduces* the cost of developing and building a television. The reason is a property of manufacturing - *economy of scale*. If every embedded system was composed of custom-made “application specific” components, such as a clockwork timer that could only fit within a specific model of toaster, the design and build costs would be very high. Each component would have to be designed for the new model.

While each embedded system has a dedicated function, very few parts of a typical embedded system are truly application specific. Some components such as *central processing units* (CPUs) [123] and memory are found in all embedded systems. Many other components can be found in all embedded systems of a particular class: for example, MPEG-2 video decoders may be found in both DVD players and digital satellite receivers.

Each of these components must be customisable for a range of applications. A CD player and a washing machine may include the same type of CPU, but the functions required are very different. Therefore, a different *configuration* is applied by the embedded system manufacturer, most commonly in the form of a *program*. A program is *software*: it specifies the actions to be carried out by a processor. This flexibility allows one device to be reused for many different applications. A generic component can be mass produced, then bought “off the shelf” and configured for use in an embedded system with any purpose: a CD player, washing machine, or television. All that is required is the appropriate software.

Software effectively creates the *functionality* of an embedded system. It is possible to translate a program into hardware, but this does not scale to a program of any size because of the need to place and route interconnections between program components [271]. On the other hand, software does scale well: the limit on a program's size is generally only imposed by the size of the computer's memory.

But software has a disadvantage. It is *sequential*, meaning that operations always take place in a defined order. This raises a second problem: *how can the speed of software operations be optimised?*

Many clever techniques have been applied to solve this problem [228, 144, 85, 236, 211, 101, 168, 103, 240, 139], with a great deal of success in reducing the *average* execution time of a program. CPUs for embedded systems can make use of some of these techniques to run programs faster.

However, this extra speed comes with a cost. When certain CPU features are enabled [167, 121, 275], it is difficult to calculate the longest time required to run a program (or part of a program). This is important when the program has to interact with external components: the program may have to respond in *real-time* to an event from elsewhere, perhaps turning off a washing machine water heater before the temperature gets too high, or ejecting some toast before it burns.

In general, this problem is expressed in terms of *deadlines*. A program with deadlines that must be met is said to be *real-time*, because it must respond to real world events in a timely fashion [43]. Such requirements introduce many issues, touching virtually all aspects of system design. Thus, the system is known as a *real-time system* (RTS), and is further described as *hard real-time* if deadlines are strict.

Because some CPU features can make program timing analysis difficult, it is not always easy to be sure whether a program running on certain types of CPU can *safely* be used as part of a larger embedded system. A hard real-time system can only be described as safe if deadlines are always met. So there is a third problem: *how can speed be increased while preserving safety?*

Methods exist to determine whether a program will meet its deadline when running in a particular environment [205]. These operate by computing the *worst case execution time* (WCET) of the program: the longest period of time that could possibly be required to run the program, in the worst possible input conditions. Sophisticated methods exist to perform this *timing analysis* [192, 159, 207, 34, 280], but it is not always easy to apply these to a particular CPU or program. Simply put, CPUs are very complex, and this makes them difficult to model.

A variety of solutions have been proposed for this problem, including CPUs that facilitate timing analysis [9, 69, 217], application specific hardware [20, 215], new execution paradigms [204, 61], and using reconfigurable hardware [260]. All of these solutions are interesting, and all are focused on a single problem: *what is the best way to build an architecture to make timing analysis easy?*

In this work, it is assumed that an embedded system is used to implement some group of software tasks that have hard deadlines. Given this assumption, the work examines possible answers to the problems identified above, specifically:

- How can the speed of real-time software operations be optimised?
- How can speed be increased while preserving safety?
- What is the best way to build an architecture to make timing analysis easy?

This is done by looking at the properties of architectures that could provide answers to these problems, and at the requirements and capabilities of WCET analysis tools. This information is used to propose, specify, implement and test a new architecture for embedded hard real-time systems. This architecture is then applied to the problem of increasing execution speed in a predictable fashion.

1.1 Thesis Aims

This work aims firstly to explore the architectural issues relating to the problems outlined above. Next, a new CPU architecture for embedded hard real-time systems will be characterised, prototyped and implemented. Finally, this architecture will be evaluated as a possible solution to the problems.

The new CPU architecture will be part of the recent trend towards architectures that facilitate timing analysis [69, 9, 204, 217] for hard real-time systems. It will extend existing work to move closer to the ultimate goal of that trend: a scalable CPU architecture that enables program WCETs to be (a) estimated accurately, and (b) reduced. To do that, the following subgoals must be satisfied by the new CPU architecture:

- **Amenability to timing analysis.**

The CPU architecture will be designed to support the best methods of timing analysis found in previous work. It will only be possible to consider an architectural feature if it supports such analysis.

- **Reduction of worst case execution time.**

Simple CPU architectures support timing analysis very well, but lack the performance required to support some real-time applications. Conventional approaches only improve average performance. This might not improve the worst case performance at all, and it might make the worst case performance more difficult to quantify. So a different approach is needed for hard real-time systems design, as an architectural feature has only provided an improvement if the WCET is reduced as a result of its presence.

- **Structural scalability.**

Structural scalability is a property defined [39] as:

“a system...[is]...structurally scalable if its implementation or standards do not impede the growth of the number of objects it encompasses, or at least will not do so within a chosen time frame.”


Some platforms for hard real-time systems do not meet this requirement. For example, the space available for “hardware threads” in [1] is limited by physical hardware space. And while [129] can support an unlimited number of different co-processors through run-time reconfiguration of hardware, the co-processors are non-relocatable and strictly limited in size. The new CPU architecture must avoid putting restrictions on the number of tasks or the size of tasks to avoid being limited to systems of a certain complexity.

1.2 Digital Appendix

This work comes with a companion data archive, containing software, hardware designs and experiments written to support the work. Documentation for the archive is provided in Appendix A, but references to specific programs and files can also be found throughout this document. These references take the form shown below:

Some of the files in the archive

- ▷ /README.txt
Documentation file, in the root directory of the archive.
- ▷ /install.sh
Installation program.
- ▷ /mcgrepl
Support files for chapter 5.



1.3 Thesis Structure

The layout of the thesis is as follows. This chapter sets out the objectives. Chapter 2 looks at related work, then chapter 3 describes the motivation and hypothesis for the work. Chapter 4 sets out a proposed architecture as an answer to the problems listed here, which are followed in chapters 5 and 6 by two architecture implementations: first a static prototype, then an extensible refinement. Chapter 7 describes a WCET reduction algorithm that makes use of the proposed architecture, and chapter 8 evaluates it against previous work. Finally, chapter 9 concludes the work with a summary of the findings.

Chapter 2

Literature

This work aims to characterise and implement a CPU architecture for embedded hard real-time systems, meeting the requirements defined in section 1.1. This literature review is divided into two major parts:

In sections 2.1 to 2.3, research in the field of worst case execution time analysis is reviewed. The problems caused by complex CPU architectures are examined, and previous solutions are described. The aim of this part is to provide an understanding of issues related to the requirement for amenability to timing analysis.

In sections 2.4 to 2.6, various methods that could be used to improve the worst case performance of embedded systems are reviewed. The aim of this part is to provide an understanding of the performance improvement methods that may be used when reducing worst case execution time is the issue, and amenability to timing analysis and scalability are also requirements.

2.1 About Timing Analysis

In section 1.1, the requirements state that the CPU architecture must be amenable to timing analysis.

Timing analysis is an essential part of guaranteeing the *schedulability* of a hard real-time system. Consider a hard real-time system as a program composed of one or more *tasks*. Given this commonly-used model, schedulability analysis is [214]:

“...the process of deciding whether or not a scheduler can schedule a particular set of tasks without missing any deadlines.”

Schedulability analysis is carried out using information about the CPU time required by each task. Because it is important to guarantee that deadlines are met in the worst possible case, the worst possible time requirement is used for each task. This time is known as the *worst case execution time* (WCET), and it is determined by analysis for a particular task and a particular CPU architecture. According to Puschner and Burns [205]:

“WCET analysis computes upper bounds for the execution times of pieces of code for a given application, where the execution time of a piece of code is defined as the time it takes the processor to execute that piece of code.”

In the general case, exact schedulability analysis is intractable [46]. However, it is possible for specific cases, such as models that involve only *periodic tasks* [141] (which execute with an exact frequency), and models that use server tasks to isolate aperiodic activities [43]. Schedulability

analysis is a part of software design, and it is therefore outside of the scope of this work. However, the requirements of schedulability analysis are important in the sense that they partly define the properties of effective WCET analysis tools.

WCET estimates are often used because computing an exact WCET value for a general program is difficult for reasons that are discussed in the following sections. However, estimates can be very close to the actual WCET value. Estimates are required to be [205]:

- **Safe:** greater than or equal to the actual worst execution time for the task,
- **Tight:** as close as possible to the actual worst execution time for the task.

The first requirement ensures that schedulability analysis will always provide each hard real-time task with sufficient time to complete its work before its deadline. The second requirement aims to minimise this time allocation, so that the CPU resources can be used efficiently by other tasks.

WCET estimates are dependent on both the application and the hardware it runs on, including the CPU and memory system, because all hardware components can affect execution time. WCET analysis tools conventionally assume that the application will not be interrupted by other applications or operating system routines [205], because the timing effects of such *interference* are unbounded.

2.1.1 WCET Estimation

A naïve approach for WCET analysis would involve simple execution with different data sets. However, in the general case, it is not possible to tell if a measured execution time really represents the worst case. Nor is it possible to evaluate the time for *all* possible input data by running the program with each possible input (an “*explicit path enumeration*” approach), because that would require at least $O(2^n)$ operations for n bits of input.

However, it is possible to directly compute the WCET of small sections of a program. At the lowest level, programs are composed of *machine instructions*: primitive operations that are interpreted by a CPU. In any sufficiently simple CPU, the execution time of each instruction is either a fixed quantity (e.g. 1 clock cycle), or dependent on some external constant such as the latency of the memory. WCET analysis is simple at this level because the instruction execution time is not dependent on inputs. The problem remains simple for any block of code that is not dependent on either execution history or external input. One example of part of a program with these properties is called a *basic block*: a sequence of instructions containing exactly one branch instruction, located at the end.

WCET analysis combines low level measurements of block execution times with high level path information. The analysis problem spans three domains as shown in Figure 2.1: source code (which provides the path information), machine instructions (which are used to calculate execution times), and a translation step to move information between the preceding two.

The WCET value is often an estimate rather than an exact value because automatically enumerating all possible paths through the program is not possible in general [205]. For example, calculating the number of iterations needed to complete a loop is undecidable [159] (this is the *halting problem*). This is why the high level source code must be consulted for information about execution paths. WCET analysis tools rely on the user to explicitly indicate path information, such as the maximum number of iterations of a loop. This is often done using annotations embedded in source code [206, 50] or separate source files [192]: effectively, the annotations are constraints

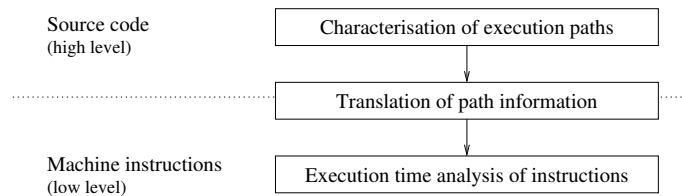


Figure 2.1: The three problem domains for worst case execution time analysis.

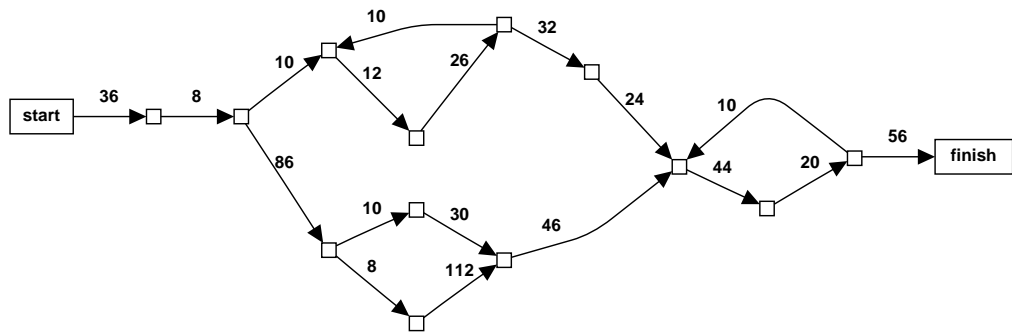


Figure 2.2: An execution path graph for a simple program, based on an example from [207]. The graph edges represent basic blocks, and each is labelled with the time cost of executing it. The units could be any measurement of time: CPU clock cycles is common. Nodes represent branch points, where execution flow may take one of two or more directions, and join points, where multiple flows lead to the same block. The program contains two loops: the maximum number of iterations of each of these will need to be bounded by some sort of constraint.

on program behaviour. A safe WCET estimate for a program may be produced by the following basic process [205]:

- Compilation of the program [2] to obtain machine instructions from source code.
- Characterisation of execution paths - generate a graph of execution paths through the program, based on high level language semantics, annotations, and other constraints. A sample graph for a simple program is shown in Figure 2.2.
- Translation of path information - obtain the machine code for each edge of the graph.
- Execution time analysis - the WCET of each edge is computed using the machine code. Each edge is labelled with its WCET.
- Recursively collapse the graph using simple rules that preserve safety [28]. These rules merge subgraphs into single edges, with a new WCET:
 1. The WCET of a sequence of edges is the sum of the WCETs of each individual edge.
 2. The WCET of a conditional branch (e.g. `if`) is the WCET of the most expensive branch direction, plus the overhead for evaluating the condition.

```
if ( x < 0 ) {
    condition = True ;    /* block (1) execution time 55 */
    y = x / 5 ;
} else {
    condition = False ;  /* block (2) execution time 10 */
    y = x + 4 ;
}
if ( condition ) {
    z = y + 2 ;          /* block (3) execution time 5 */
} else {
    z = y / 9 ;         /* block (4) execution time 50 */
}
```

Figure 2.3: This example illustrates two infeasible paths. If statement (1) is executed, statement (4) cannot be executed. Similarly, if statement (2) is executed, statement (3) cannot be executed.

3. The WCET of a loop is the maximum WCET of the loop contents, multiplied by the maximum number of iterations, according to the source code.

The result of this process is a graph with one node, labelled with the WCET estimate for the entire program.

This simple WCET analysis method will produce a value that is guaranteed to be safe, provided that the edge WCETs have been computed correctly, but there is no guarantee of tightness. The WCET estimate may be very pessimistic, i.e. much larger than the actual WCET. This is because it is not possible to incorporate all of the constraints that affect execution into the analysis, so more elaborate methods are usually required.

2.1.2 Reducing Pessimism in WCET analysis

WCET estimates need to be as close as possible to the actual WCET so that CPU resources can be allocated efficiently. The problem of reducing the overall pessimism of the WCET analysis process is the subject of ongoing research.

One source of pessimism is inadequate modelling of execution path constraints. Bounding loops and recursion makes safe WCET estimation possible, but it does not provide information about *infeasible paths*. A simple example is provided in Figure 2.3. Given this example, a simplistic WCET analyser will examine each `if` statement in isolation. This produces a safe but pessimistic WCET estimate of $\max(55, 10) + \max(5, 50) = 55 + 50 = 105$, using the execution times of blocks (1) and (4). However, no path through (1) passes through (4): the only possible paths are [(1), (3)] and [(2), (4)]. The actual WCET is $\max(55 + 5, 10 + 50) = 60$.

This can be avoided to some extent by analysis of high level source code: Altenbernd [4] describes symbolic execution of source code to store information about constraints that are known to hold along possible execution paths. For example, the constraints set by block (1) in Figure 2.3 eliminate block (4) from consideration.

Generally, symbolic execution is called *abstract interpretation* (AI). The benefits of AI go beyond detecting infeasible paths: AI can also be used to derive other execution constraints in some

Constraint	Effect
loop L [1,10] times; samepath(A,C);	Loop L repeats at least once and at most 10 times. When statement A executes, statement C always executes, and vice versa.
(not A) imply loop L 10 times; execute A [0, 1] times inside L;	If A is not executed, loop L is repeats exactly 10 times. During execution of L, statement A will execute at most once.

Table 2.1: Examples of IDL statements from [192].

```

procedure A ( N : POSITIVE ) is
begin
  --{ loopcount (1,N);
  for I in POSITIVE range 1..N loop
    --# assert TRUE;
    --{ loopcount (I,N);
    for J in POSITIVE range I..N loop
      --# assert TRUE;
      S;
    end loop;
  end loop;
end A;

```

Figure 2.4: Annotations in a procedure indicate the bound on the total number of loop iterations, from [49] page 101. These allow a WCET analysis tool to calculate that statement *S* will only execute $\frac{1}{2}n(n-1)$ times, rather than n^2 times which would be assumed if the bounds on both loops were fixed. Two types of annotation are seen here: conventional SPARK/Ada annotations (`--#`) and timing annotations (`--{`) as proposed in [50].

cases, including loop bounds. In [78], Ermedahl and Gustafsson describe a method for finding all constraints within programs written in a subset of C or Smalltalk. Their approach runs the program but considers all variables as a feasible range of values rather than a single value. This can be used to discover some loop bounds and infeasible paths automatically, independently of the input.

However, these approaches will not work in all cases since deriving constraints is equivalent to the halting problem in general. Therefore, ways to specify information about loops and infeasible paths are always needed. In [192], Park describes the use of a constraint language (“IDL”) with powerful features for expressing the relationships between statements. Some examples are illustrated in Table 2.1. These constraints are placed in a separate source file and linked to program source files via labels attached to statements. Using this model, constraints can be easily applied to any subset of all statements, regardless of their scope or encompassing function.

An alternate (but essentially equivalent) approach is to place these annotations directly in source code. In [50], Chapman et al. describe extensions to the SPARK/Ada language for hard real-time systems. At the time, SPARK already included an annotation mechanism which was used to prove program correctness [49]. Chapman extended this mechanism to support new timing annotations that are similar to the existing SPARK syntax. A sample annotated procedure is shown in Figure 2.4.

Through path constraints of this type, it is possible to express all possible program behaviour,

Variable	Meaning
$f(x)$	Upper bound on number of executions of basic block x , given all constraints. Computed by integer linear programming algorithm.
$\gamma(x)$	Execution time of basic block x . Constant.
Z_G	Program WCET, defined by equation 2.1.

Table 2.2: Variables specified by the integer linear programming model of Puschner and Schedl [207]. The integer linear program selects the values of $f(x)$ that maximise Z_G .

not just infeasible paths. The *implicit path enumeration technique* (IPET) approach to WCET analysis proposed by Li and Malik [159] applies these constraints to an execution graph by expressing them as inequalities. For example, if a statement n is executed $f(n)$ times, then the constraint that statement A executes once for every five executions of B is expressed as $5f(A) = f(B)$.

For IPET, the WCET Z_G of an execution path graph $G = (V, E)$ is defined by this equation:

$$Z_G = \sum_{x \in E} \gamma(x)f(x) \quad (2.1)$$

where $\gamma(x)$ is the time cost of executing a basic block x , and $f(x)$ is the number of times that basic block will be executed in the worst case. Each $\gamma(x)$ is computed using a CPU model and has a constant positive integer value. Each $f(x)$ is computed by using an *integer linear program* to calculate the values of $f(x)$ that maximise Z_G . This is possible because the value of $f(x)$ for a basic block x is constrained by f for x 's successors and predecessors, and by constraints specifying infeasible paths, loop bounds, and other execution bounds. Through IPET, the WCET analysis problem is mapped onto a form of constraint problem. An example of the IPET process is given in Appendix E.

Puschner and Schedl [207] have shown that the exact WCET value for a program can be computed if the complete set of path constraints for the program are specified. (A second implicit assumption is that $\gamma(x)$ is constant for each basic block x .) The proof is based on a formal model using the same approach as Li and Malik. In this model, the total number of executions of each piece of code is expressed as a parameter of the circulation problem from graph theory [63]. The program is expressed as a *timing graph* (T-graph) in which every edge x is a piece of non-branching code, labelled with a known (constant) execution time ($\gamma(x)$) and a ‘‘flow value’’ ($f(x)$) that represents the upper bound on the number of executions of the associated code. Nodes represent branches and join points, and each has the property that $\sum f$ for all incoming edges is equal to $\sum f$ for all outgoing branches. Table 2.2 lists the variables involved. Additional bounds on f are applied using path constraints like the ones listed in Table 2.1.

The model is then transformed to a series of inequalities that describe the graph structure and all the constraints that apply to it. Like the inequalities used in the Li and Malik model, these are solved as an integer linear programming problem, which computes worst case values for the number of executions of each edge (i.e. f) such that Z_G is maximised.

Integer linear programming is an elegant way to combine structural information about a program with path constraints. It allows infeasible paths to be eliminated, handles bounds on loops and recursion, and is capable of supporting complex constraints such as non-rectangular loops (e.g. where an outer loop executes once for each x from 1 to 100, and an inner loop executes once for

each y from 1 to x). Path constraints can even be used in the manner of procedure parameters to propagate constraint information from callers to callees - a symbolic annotation approach for this purpose is proposed by Bernat and Burns [33].

A potential disadvantage of the approach is that the integer linear programming problem is NP-hard. Algorithms to solve it require $O(2^n)$ steps in general. (The topic of algorithmic complexity is revisited in section 2.4.1.) However, Li [159] observed that when the path constraints are restricted to those available in IDL (e.g. those in Table 2.1), “the integer linear programming problem is equivalent to a network flow problem, which can be solved in polynomial time.”

2.1.3 The Translation Step

Modelling the structure of a program is only one part of the task of WCET analysis. Low level information about the timing of each basic block of program code is also required. As Figure 2.1 illustrates, a translation step is needed to manage the relationship between high level execution path information and low level machine instructions.

Conventionally, machine instructions are produced from source code by a compiler [2]. Early compilers maintained a straightforward relationship between high level statements and machine code. With such a compiler, translating information between the high and low level domains is very simple, especially as the relationship between source code and machine code is already maintained for debugging purposes.

However, this type of machine code does not use CPU resources efficiently. To improve performance, compilers generally apply *optimisation* algorithms which preserve the *meaning* of code but change the low level structure so that CPU resources are used more effectively. (The effects of software optimisation are described in section 2.4.2.)

Optimisations make the translation step non-trivial [205] as structural information is partly lost. Two classes of translation approach have been demonstrated to handle this problem:

1. **Invasive** approach: the compiler transforms execution path information at the same time as optimising code. This approach requires modifications to compiler source code, hence *invasive*. Compilers that have been extended with this feature include `vpo` (in [183]), `gnat/gcc` (in [169]), and `lcc` (in [162]).
2. **Co-transformation** approach: optimisations are applied outside the compiler by a second tool. This approach is demonstrated by Engblom [76], who proposes *optimisation description language* (ODL) as an input to a tool which applies optimisations and outputs information to match high level path information with low level machine instructions.

These approaches make it possible to carry out WCET analysis on optimised programs. This is essential for efficient use of a CPU.

2.1.4 CPU Modelling

The final problem domain of WCET analysis is the low level model used to calculate the execution time of pieces of code. High level WCET approaches have made the assumption that the execution time of each piece of code is constant, or at least outside the scope of discussion [207, 192, 159]. These assumptions hold for simple CPUs such as the Motorola 68000, where execution time is constant for each instruction [181], but are invalid for more modern CPUs where a number of features can cause the execution time of the same basic block to vary.

CPU design is a complex topic, but it is important to explain the nature of these features and the reasons for their introduction. This is an essential part of the work, because the requirements force an understanding of why state of the art CPUs are not amenable to timing analysis. If the state of the art is to be advanced, it is necessary to understand the ways in which current approaches are lacking. Therefore, discussion of WCET analysis is suspended at this point to make way for a digression into CPU technology. In section 2.3, the review of WCET analysis continues with an examination of the techniques that have been applied to account for modern CPU features.

2.2 A Brief History of CPU Technology

In the early days of computing, CPUs had to be built with minimal hardware. Electronics were bulky and expensive, there was no design automation, and components were unreliable [83]. The machine instructions understood by a CPU (the *instruction set architecture*, or ISA) had to be simple to keep construction and maintenance costs low. The invention of the *integrated circuit* (IC or *microchip*) enabled advanced CPUs to be mass-produced. The first complete CPU on a chip (*microprocessor*) was the Intel 4004 [137], built in 1971. Before this, CPUs had been built using multiple chips or discrete components, at a higher cost.

Since that time, economic forces have driven development of CPUs with higher performance. The performance (operations per unit of time) of a CPU may be measured in *millions of instructions per second* (MIPS), but the use of standardised benchmark programs [108, 154, 70] is often preferred because MIPS values are not comparable between CPU types and do not indicate the number of operations actually completed by each instruction. Benchmarks measure the time taken to accomplish a particular task, providing a metric for overall CPU performance that is independent of low level details such as the ISA.

Note that it is important to distinguish between architectural features that provide improved *average* performance and those that provide *guarantees* about improvement. Many performance improvement mechanisms are not well suited to providing guarantees about timing: they are very effective at improving performance, but only in the “most likely” cases. In other words, they reduce the *average case execution time* (ACET) of typical programs, but they do not make any guarantees about WCET. One characteristic of such CPUs is that the execution time of any particular basic block depends on what has been previously executed. CPUs with this history dependence property are said to lack *basic block timing invariance*.

In general, performance improvements are limited by engineering issues. Two such “bottle-necks” are given particular consideration here: the memory bottleneck and the instruction rate bottleneck. These have been found to force the use of non-trivial solutions in order to achieve faster execution, and many of these only reduce ACET. This section describes the nature of each bottleneck, and successful solutions applied to avoid it. Other bottlenecks (not considered here) include limitations on transistor size, clock frequency and heat dissipation.

2.2.1 The Memory Bottleneck

CPU core frequencies increased by many orders of magnitude between the early days of ICs and the time of writing (Figure 2.5). The frequency increases have been enabled by improvements in chip manufacturing technology, allowing smaller transistors to be used.

However, the speed of external connections (particularly to memory devices) has not scaled so

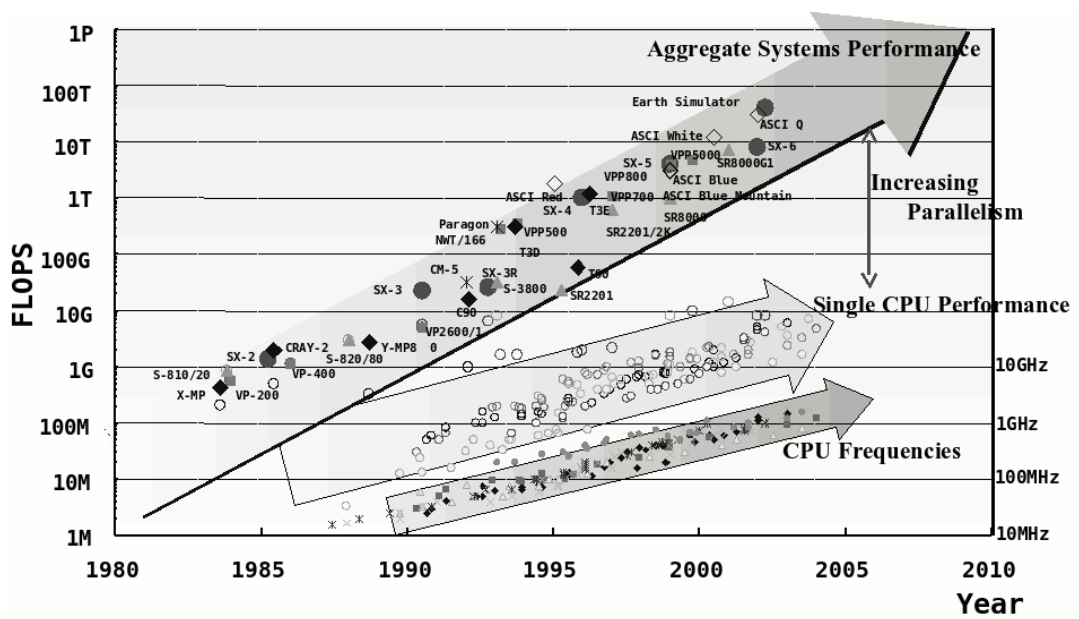


Figure 2.5: The increase in CPU core frequencies from 1980 to 2005. This chart is from [64]. FLOPS is *floating point operations per second*: in this context, this figure is analogous to MIPS.

well. Today (2008), a CPU core may operate at over 2GHz, but the memory bus clock frequency is still limited to around 400MHz. Although these frequencies do not indicate maximum memory *bandwidth* (the amount of data that can be transferred per unit time), minimum memory *latency* (the amount of time required to set up each memory transaction), or the maximum number of instructions that a CPU can execute per unit time, they do reflect the disparity between memory speed and CPU speed in modern computers.

This bottleneck has been well known for some time. It is due to the difficulty of moving data at high speed across a long distance. At very high speeds (e.g. 2GHz), even a few centimetres becomes a problematic distance. The speed of the transfer is affected by the capacitance and impedance of the transmission line, which can result in problematic transmission artifacts such as resonance.

An early solution was separate instruction and data memory, known as a *Harvard architecture*, which doubled the memory access bandwidth and allowed the size of each memory to be tailored to application requirements. This memory arrangement is named after an early implementation in the Harvard Mark I computer [131].

The average performance of memory accesses (and program ACETs) was improved by the invention of the *cache*. A cache memory unit is a small fast memory unit that sits between a CPU and a larger memory bank with slower access (e.g. access via a bus) [99]. It intercepts all memory transactions, and tries to satisfy each without sending any requests to the larger memory bank. This is done by keeping copies of data that recently passed through the cache. Whenever the CPU tries to load data, the address is compared against the addresses of the copies in the cache (called the *tags*). If the requested data is already in cache due to a recent load or store, the copy is supplied without accessing the slow large memory. If it is not in cache, an access to the larger memory is forced.

Caches reduce ACET because programs often repeatedly access the same areas of memory:

hotspots. However, it is sometimes impossible to know whether instructions or data are in cache or not, and therefore impossible to know how long an access will take. In general, caches provide only *qualified* guarantees about their contents at each position in a program, because cache operations are dependent on the execution history. However, this may still be enough to support a WCET analysis process [183].

Caches are often described as n -way associative [195]. In a 1-way associative or *direct-mapped cache*, a real memory address A is mapped onto a line L in a cache of size C by a modulo operation $L = \text{mod}(A, C)$. In this arrangement, data at address A will *conflict* with data at address $A + C$ and (in general) $A + mC$ for an integer $m \neq 0$: i.e. data with address $A + mC$ will disappear from cache as soon as A is accessed. Because C is much smaller than the entire memory of the computer, this situation is common in direct-mapped caches: for example, the machine instructions making up one part of the program can conflict with another part, particularly when function calls are being made. The effect is dependent on addresses assigned to machine code by compiler tools.

To reduce this effect, *multi-way associative* caches are used: n -way associative with $n > 1$. These store n pieces of data for each cache line. For example, a 2-way associative cache can store data for address A and $A + C$ on the same line. The chance of a cache conflict is reduced by this scheme, as noted in [44]. An extreme form of the scheme is to make the cache *fully associative*, so $C = 1$ and n is the size of the entire cache. In this scheme, data from any address can be placed anywhere in the cache. The most commonly implemented scheme for selecting a data element for replacement is selection of the *least recently used* (LRU) element [195], but random replacement and “first-in first-out” schemes have also been considered.

Complexity is increased by the process of n -way comparison, and the need to choose an element for replacement whenever a cache miss occurs. This complexity increases hardware costs and energy consumption [139], and it makes WCET analysis more difficult because a more complex model is needed [121]. All of these factors support the argument in favour of simpler direct-mapped caching logic put forward by Hill in [125].

Multi-level caches [99] put several caches in series. These are conventionally assigned a level number to indicate their distance from the CPU. Level zero (L0) is closest to the CPU: this cache is the smallest and fastest of the caches. Other caches (L1, L2, etc.) are progressively larger, slower, physically further from the CPU, and perhaps shared between multiple CPU cores. This hierarchical scheme is an efficient way to minimise slow memory accesses in the average case.

Caches also enable *burst memory accesses*, where more than one item of memory is read at once. Burst memory accesses make more efficient use of high speed bus bandwidth, as the time cost of setup is amortised across a large number of transactions. Burst accesses usually fill an entire *line* of cache - the largest unit for data storage in the cache.

Conventional caches store instructions or data in their original form. A *trace cache* [221, 220] stores the schedule of decoded micro-operations derived from CPU instructions. The micro-operations represent low-level steps to be taken by the CPU to execute each instruction. In some CPUs, such as the Pentium 4 [135], a trace cache allows the reuse of the execution pattern for a particular piece of code. This is particularly useful when that pattern is not just a trivial copy of the original program. For example, the trace cache contents include implicit assumptions about which branches are most likely to be taken.

Caches are not the only way to minimise slow memory accesses. A *scratchpad memory* has been proposed as a way to reduce energy consumption [240, 241] and improve average performance [139]. Scratchpad memories can completely replace caches, and do not include any tag

Instruction	Purpose
IDIV	Integer division
LOOPNZ	Looping: decrement, test and jump
SCASB	Compare byte string
XLAT	Table lookup

Table 2.3: Example CISC-style instructions on Intel 8086 CPUs [230]. Source [188].

logic because they are explicitly updated by a program. An additional advantage of a scratchpad is that it responds to all memory requests in a statically predictable fashion, making scratchpads well suited as components in an RTS [244, 202, 273].

2.2.2 The Instruction Rate Bottleneck

Early computers had small memories, linked to the CPU by low-bandwidth connections, so high performance was a matter of doing more CPU operations per instruction. In this era, programs were frequently written in assembly code, so complex instructions were included in the CPU to carry out common assembly operations. Assembly programmers used these to save execution time and space, a trend which was repeated in the early microcomputer era. Some examples of these instructions for the 8086 [230] processor are listed in Table 2.3. Each has a function that could be replicated with a number of simpler *primitive* (as opposed to *complex*) CPU instructions.

These are classic examples of *complex instruction set computing* (CISC) instructions, as each involves multiple internal steps and can be replicated using two or more other instructions. CISC instructions are compact, reducing demands on memory size and bandwidth. Using CISC instructions can reduce both the ACET and the WCET of a program, because internal steps within CISC instructions can be executed in parallel, and fewer instructions need to be fetched from memory. Well-known CISC CPUs include the x86 family, the System 360 [7] CPUs and the Motorola 68000 [181]. On classical CISC CPUs, such as the original 68000, the time taken to run each instruction is independent of the CPU state, so replacing common groups of instructions with a single CISC operation reduces WCET by a predictable bound.

CISC instructions cannot always be used by C compilers because of the difficulty of recognising where code specified at a high level can be mapped onto a particular low level instruction. While division instructions (e.g. `IDIV`) are recognised and used because they map onto language primitives (e.g. `/` in C), more unusual instructions like `XLAT` (Table 2.3) are not recognised by `gcc`. The only way to use these CPU features to reduce WCET or ACET is to use assembly code, which is not practical for large programs or when portability is required. In CISC CPU design, static assumptions are made about likely sequences of instructions, and if these assumptions are not met by any particular program, the extra hardware required to execute those sequences of instructions is unused.

As the memory bandwidth problem of early CPUs was solved to some extent by caching [99], and the memory size problem was solved by advances in memory technology, less compact *restricted instruction set computing* (RISC) instructions became more desirable. RISC architectures were originally intended to closely match the capabilities of compilers [228]: the instructions all map easily to language primitives. After the high performance demonstrated by early RISC

<p>(a)</p> <pre> loop: (0) l.lbz r17,0x0(r4) (1) l.srli r9,r5,0x8 (0) l.addi r6,r6,0xffffffff (1) l.xor r15,r5,r17 (2) l.addi r4,r4,0x1 (2) l.andi r13,r15,0xff (1) l.sfnei r6,0x0 (3) l.slli r11,r13,0x2 (4) l.add r5,r11,r8 (5) l.lwz r7,0x0(r5) (5) l.bf loop (0) l.xor r5,r7,r9 exit: ... </pre>	<p>(b)</p> <pre> (0) { l.lbz r17, 0x0(r4), l.addi r6, r6, 0xffffffff, l.xor r5, r7, r9 if flag } (1) { l.srli r9, r5, 8, l.xor r15, r5, r17, l.sfnei flag, r6, 0 } (2) { l.addi r4, r4, 1, l.andi r13, r15, 0xff } (3) { l.slli r11, r13, 0x2 } (4) { l.add r5, r11, r8 } (5) { l.lwz r7, 0x0(r5), branch to (0) if flag } </pre>
--	---

Figure 2.6: Example of instruction-level parallelism (ILP) in a CRC-32 checksum loop written using ORBIS32 machine code (Appendix C, [151]). (a) shows the original code. (b) shows the code, rearranged to parallelise it into 6 groups of machine operations (0..5).

projects [144], which exceeded contemporary CISC designs even when implemented using a much older integrated circuit process, the CPU industry began to move to RISC-based designs. Through the use of an instruction cache, a RISC CPU can effectively adapt dynamically to the sequences of instructions currently in use, reducing ACET and arguably making more effective use of chip area than a classical CISC CPU which never adapts to code.

Another advantage of RISC designs was simplification of *pipelining*, where CPU tasks are broken down into stages, such as *decode* and *execute*. The pipeline used in the “MIPS” RISC processor is described by [123]: it enables higher core frequencies by reducing the propagation delays between pipeline stages. Several different instructions can be in the various stages of execution at a once.

The original MIPS pipeline did not allow instructions to execute in parallel, or in any order other than the one specified by the compiler or programmer: it was a *single issue* pipeline. More advanced pipelines reduce ACET with *multi issue* technology, where instructions are executed in parallel if possible - *instruction level parallelism* (ILP). The ILP present in a simple hotspot is shown in Figure 2.6.

Generally, greater ILP is possible when the list of instructions to be rearranged is longer. In most code, this “fine-grained parallelism” is limited by control transfers at the ends of basic blocks [144], because the subsequent basic block is not known until the exit condition of the basic block can be evaluated. The limit on ILP under these conditions is approximately two parallel instructions in most programs [186, 268].

However, this problem can be circumvented by temporarily ignoring conditional control flow and executing subsequent instructions *speculatively*. If speculation is based on perfect predictions of the future, greater parallelism is possible [186], as more ILP can be obtained from the code [56]. In practice, speculation has to be based on data from execution history, and is not completely accurate.

This use of historical data is a more general case of the problem with a cache, because the operation of a CPU component depends on its previous inputs. The history dependence means that basic block execution times may change. A typical CPU will aim to ensure that the average

execution time is minimised, but nevertheless, the maximum execution time can be much larger than this average when certain types of dependence exist [167, 275].

Several mechanisms for speculative execution are described below, along with more advanced techniques for improving average code performance such as vectorisation and dynamic recompilation.

1. **Superscalar, In order:** the instruction stream is a sequence of CISC or RISC operations. The CPU attempts to execute operations in parallel whenever possible, but never attempts to reorder the operations. This allows some ILP to be exploited. The pipeline complexity is increased somewhat, with additional functional units, but since operations are never reordered, register access hazards are avoided and exception handling is no more complex than in a classical design.
2. **Superscalar, Out-of-order:** the instruction stream is a sequence of CISC or RISC operations. The CPU looks ahead within this stream and attempts to execute operations as early as possible by finding instructions with input operands that are available. For example, the first three operations from Figure 2.6(a) can execute simultaneously, because there is no data dependence between them. In general, greater ILP can be exploited by an out-of-order CPU because a larger set of instructions are available for issue in every clock cycle.

The hardware needed to manage out-of-order execution (an *out-of-order issue unit*) is complex. Two classic schemes are a scoreboard approach [31] for detecting data dependence conflicts and waiting until they are resolved (*stalling*), and Tomasulo's algorithm [254], which attempts to resolve data dependences by renaming registers. The topic is now mature: an approach leading to an efficient hardware implementation is described by Sohi [236], which combines aspects of both scoreboard and Tomasulo's algorithms.

All of these schemes assign operations to execution units as soon as possible. As execution units may take multiple clock cycles to complete an operation, the set of units that are busy when execution of a basic block begins is dependent on execution history.

Speculative execution is effectively a requirement for out-of-order issue. Control dependences are created by conditional branches, which mark the ends of basic blocks. After these branches, it is not known which instruction will be executed next until the branch condition is evaluated. In order to avoid this limitation, dynamic speculation is used by the out-of-order issue unit. Branch prediction is used to guess whether a branch will be taken or not. This is usable because branch prediction mechanisms [89] are often correct [124, 233] because predictions are based on dynamic information from recent history. If no branch prediction is used, the CPU must wait for the branch condition to be known, which may result in stalling. If reducing ACET is a design goal, then it is preferable to make a prediction and then undo the effects of speculation in the unlikely event of the prediction being incorrect.

When predictions fail, the CPU has to return to the state before the branch point. All speculative executions are undone. Sohi's algorithm handles this elegantly [236], but earlier approaches exist [234]. The misprediction time penalty is unavoidable, but it is amortised by the average speed gains from speculation in most cases. On the Pentium 4 CPU, the penalty is at least 20 clock cycles [118]. Branch mispredictions are a good example of an event that can greatly increase WCET.

3. **Explicit Parallelism:** the instruction stream generated by the compiler is parallel. This approach was first used in *very long instruction word* (VLIW) processors [87], where each instruction could specify several operations to be carried out in parallel. A similar idea was adopted by Intel and HP as *explicitly parallel instruction computing* (EPIC) [223] for use in the “Itanium” IA-64 architecture [109], where each instruction specified a set of operations that could be carried out in parallel or in any other order. The improvements made by EPIC over VLIW are unclear, and have been regarded as marketing differences rather than new technologies [87]. However, EPIC does aim to address some of the problems with early VLIW ISAs: in particular, a lack of support for forward compatibility with future CPU architectures [223]. Lacking this feature would force programs to be recompiled for newer CPUs.

VLIW and EPIC compilers exploit ILP using one of two basic techniques: *acyclic scheduling* for non-looped code, and *cyclic scheduling* for looped code [87]. The most practical acyclic scheduling approaches [85, 48] work by obtaining long sequences of instructions from a program, with assumptions regarding the most likely control flow. These assumptions are often guided by profiling. The scheduler includes tests to ensure that the assumptions are correct, and handle a “misprediction” condition if they are not. Cyclic scheduling is designed for efficient mapping of *inner loops* to VLIW architectures (inner loops are loops that do not contain other loops or subroutine calls) [60, 211, 165]. The approach used is software pipelining: multiple loop iterations are overlapped to find greater ILP.

VLIW CPUs have proved an effective way to reduce program ACETs. However, instruction caches have to be used. The memory bandwidth requirements of VLIW code are generally higher than RISC or CISC code, and if a cache does not offset this requirement, the performance gains are lost. This poses difficulties for WCET analysis.

4. **Vector:** each instruction from the instruction stream is carried out on multiple data sources. This is most effective for operations that are vector quantities, such as co-ordinates or colour values, as it avoids the need to repeat a sequence of commands for each member of the vector. It is now often referred to as *single instruction, multiple data* (SIMD) processing, and is supported by recent Intel processors through Streaming SIMD Extensions (SSE) [196].

Vector processing can be extremely powerful, but it is only possible when the code is “vectorisable” [268], and then only when the CPU can detect the possibility of vectorisation. Vectorisable code applies the same operation to many data sources that can be considered independently of each other. In general, software does not have this property [268], so vector processing cannot reduce the ACET or WCET of all programs. Cyclic scheduling can be regarded as a way to partially vectorise inner loop code, but its effectiveness as a vectorisation technique is limited by inter-iteration dependences.

5. **Multi-core:** some applications can be split into tasks that can execute on physically separate CPUs, such as the IBM Cell multiprocessor [142]. These tasks are *threads* - software elements within a program that may execute simultaneously but share a memory space. Threads communicate with each other via memory and explicit synchronisation operations. The use of threads is a *multiprocessing* approach for improving performance by exploiting *task-level parallelism* (TLP), and it relies on developers to develop multithreaded applications.

Exploiting TLP does not directly reduce ACET or WCET because both are measured at the

task level. However, it should be noted that any task may affect the execution of other tasks through interference: for example, by changing the state of a cache. This may occur between tasks that run in parallel or in the sequence. Exploiting TLP can improve the average performance of a complete program, but it may also increase the WCET of individual tasks by introducing new interference effects. This issue is revisited in section 2.3.12.

6. **Simultaneous multi-threading:** multiple threads normally execute on a CPU by *time slicing*, i.e. the CPU fully switches between threads, executing exactly one at a time. In a *simultaneous multi-threaded* (SMT) architecture, the CPU elements are shared between multiple pipelines, allowing one thread to make use of idle units while another is *stalled*: that is, waiting for a pipeline stage to complete. This feature was given the trade name “hyper-threading” for the implementation in the Intel Pentium 4 [138]. In the Pentium 4, SMT allows one physical CPU to act as two “virtual” CPU cores, so task-level parallelism is exploited by a single CPU core. Like multi-core operation, SMT can improve overall performance, but interference effects may increase WCETs for each task.
7. **Dynamic Recompiling:** Superscalar CPUs carry out a very simple form of dynamic recompilation. Machine instructions are translated into internal micro-operations, which can then be rescheduled and executed in parallel. However, this recompilation has restricted functionality, partly because it is implemented as hardware. More powerful optimisations can be applied by a software compiler.

The *Dynamically Architected Instruction Set from Yorktown* (DAISY) is a virtual machine that uses dynamic recompilation to execute code for one type of CPU (e.g. PowerPC) on another (e.g. an experimental VLIW CPU) [66, 243]. DAISY thus acts as a software replacement for an out-of-order issue unit. The input machine code is recompiled on demand, then stored in a cache for faster access in the future.

The approach is similar to the use of a *just in time* (JIT) compiler to retarget Java bytecodes for a host CPU [245], but in that arrangement, JIT compiled code still passes through a final stage of recompilation in a CPU out-of-order issue unit. In DAISY, the emitted code is directly executed by a VLIW processor, without any further recompilation.

The advantages of this approach include hardware cost savings, as a simpler VLIW architecture can be used and components that maintain compatibility with legacy code (e.g. 8086) can be removed and simulated by DAISY. Additionally, the DAISY target CPU can be upgraded freely: even to other types of CPU with entirely different instruction sets. Provided that the DAISY code generator backend supports the new target, the input code will continue to work correctly. Thus, forward and backward compatibility are assured through a software translation layer.

Transmeta [255] launched a number of CPUs with DAISY-like behaviour, including Crusoe and Efficeon [256]. These execute x86 instructions by a combination of interpretation and recompilation to an internal VLIW instruction set. The CPU reserves an area of external memory for caching recompiled code and storing temporary data: this is inaccessible to programs other than the recompiler, which is always running in the form of a *hypervisor* and boots from internal ROM before any x86 code can be executed. During operation, CPU resources are shared between the recompiler and the user applications. As with DAISY, this

approach enables the CPU to execute x86 code efficiently without the high cost of out-of-order issue units and legacy support.

These technologies have been shown to reduce ACET, but it would appear that they are not suitable for WCET reduction. In order to quantify the WCET reduction and prove the safety of a task, it would be necessary to model the dynamic behaviour of the recompiler under all possible circumstances, just as it is necessary to model the dynamic behaviour of a CPU pipeline in order to be sure that the true bound on the WCET has been determined.

All of these approaches obtain higher average performance by carrying out operations in parallel. However, many of them are limited in this regard by software itself. According to [268]:

“parallelism within a basic block rarely exceeds 3 or 4 [parallel instructions] on the average.”

This motivates parallelism that is exposed by the programmer, either in the form of vectorisable code or by splitting operations into separate tasks that can execute independently on a multi-core processor. But such approaches effectively require extensions to existing software languages, which is a barrier to widespread adoption, so CPU and compiler designers have focused their efforts on maximising use of the ILP available in general code.

2.2.3 Effects of Dynamic Features

Almost all of the features described in the previous section can cause a wide variance in the execution time of a piece of code. The total time taken to execute the code depends on what the CPU was doing beforehand: the execution history. Execution time depends on the contents of instruction and data caches (and thus whether the code will need to be fetched from memory), the state of the pipeline and issue unit, the contents of the branch prediction unit, and the interactions between these components.

WCET analysis tools are forced to model this behaviour in order to produce a tight WCET estimate. In some cases this would appear to be an intractable problem, such as the general case of all possible interactions between two tasks in an SMT environment. In other cases, it would appear to be an extremely costly problem, perhaps requiring as much engineering effort as the original CPU design, if not more. In the next section, efforts to create analysis tools for modern complex CPUs are described.

2.3 WCET Analysis of a Complex CPU

The previous section identified a range of CPU features that can cause a variance in the execution time of code. Until the 1990s, such features were not widespread outside of high-performance computers, and CPUs without pipelines or caches such as the Motorola 68000 [181] were commonly used in hard real-time systems. CPUs of this type have the *basic block timing invariance* property: any particular basic block would always execute in the same number of clock cycles. For example, [181] includes an appendix of timing tables that enable the exact execution time in clock cycles to be calculated for any basic block. Early WCET analysis approaches relied on such timings being well-defined [192, 159].

As caches and pipelines were adopted by CPU manufacturers to reduce ACET, support for these features became regarded as necessary for future WCET analysis tools. The problem is that the

operation of the new features depends on execution history, invalidating assumptions made by some WCET analysis approaches. In order to account for this, a WCET analysis tool can simply assume the worst possible timing in every case, but this does not meet the tightness requirement (section 2.1): it can be very pessimistic. It is also unrealistic, because it is often possible to make *some* statement about the execution history: for example, from the graph of execution paths. However, it is not possible to do this without an accurate CPU model that predicts (a) what is known at each point in execution, and (b) how it affects the execution of the next piece of code.

2.3.1 Modelling a Pipeline in Isolation

Initial attempts to model complex CPU features began by considering pipelines and caches in isolation. In [294], Zhang et al. describe a model of the pipelined 80C188 CPU which can predict the worst-case execution time of any basic block given the simple two-stage pipeline. The model accounts for the overlap of instruction fetch and execution in the 80C188 pipeline. However, the boundary between basic blocks is considered pessimistically. Each block is assumed to start with the pipeline in the worst possible state.

Pipelines add complexity to analysis because execution of instructions is overlapped with fetching and decoding [123]. Frequently, instructions from multiple basic blocks are being processed at the same time. As some instructions take longer to process than others, it is possible that some pipeline stages will be stalled when the first instruction of a new basic block begins to execute. Therefore, the time taken to execute the new instruction depends on the contents of the pipeline.

By assuming the worst possible pipeline state at the beginning of each basic block, Zhang's model could be used with WCET analysis approaches that assumed that basic block timing was independent of execution history. But as subsequent research found [162], the model would not scale to longer pipelines such as the five-stage pipeline of classical RISC CPUs [144].

2.3.2 Modelling an Instruction Cache in Isolation

In [16], Arnold et al. consider the problem of modelling an instruction cache. To do this, an abstract model of a CPU cache is used, containing incomplete information about the cache state at any particular position in the program. This allows some instruction accesses to be identified as guaranteed cache hits by using information from the graph of execution paths (section 2.1.1).

It is necessary to use an abstract model of the cache state because using an exact representation of the cache state would force explicit exploration of all possible paths through the program. The computational complexity of this task makes it intractable in general, so an abstract representation is used with the property that it is possible to combine information from different paths in a meaningful way, assuming the worst case whenever several cases are possible.

The effects of this can be seen in a simple example. Figure 2.7(a) shows a simple loop which can fit entirely within the cache. The first instruction (A) has two possible predecessors (B and C). When the loop is entered from B, it is not necessarily in cache, so Arnold's approach assumes that each instruction fetch is a cache miss. But when the loop is entered from C, the abstract cache state indicates that the loop is certainly still in cache. Therefore, cache hits can be safely assumed. This information could also be obtained from an exact model of the cache state, but the search space required to enumerate all possible cache states would double at every branch. The use of abstract cache states avoids this problem.

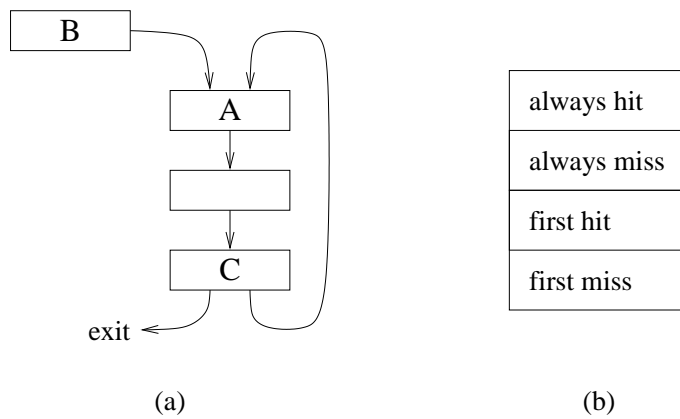


Figure 2.7: (a): Arnold’s approach accounts for situations where cache contents are known, such as loop iterations after the first. (b): The approach assigns one of these four classifications to each instruction. In example (a), instructions A and C are classified as “first miss”, indicating that a cache hit is guaranteed after the first loop iteration.

Arnold’s approach computes abstract cache states for every node of the execution path graph, making safe assumptions whenever multiple paths reach a particular node. It is able to classify each instruction according to one of four groups (Figure 2.7(b)), and can account for much more complex arrangements of code than the simple example illustrated in Figure 2.7(a). For example, the approach is able to handle nested loops and non-recursive execution of subroutines.

Recursive subroutine calls are not supported as *virtual inlining* [251] is used. Each call to a function F is represented as a distinct version of that function F_i . This is done because the effects of a function call on the cache state is dependent on the context in which the function is called. Therefore, for the purposes of abstract cache modelling, each function is considered to be an inline part of its caller. The side effect is that recursion is not possible: a cycle in the function call graph would result in infinite expansion. However, this problem can be avoided [183] as recursive functions can be represented by loops.

One criticism [162] of Arnold’s approach is its pessimistic handling of potential cache conflicts (section 2.2.1). When a conflict is possible, a conservative assumption is always made. For example, consider the loop in Figure 2.8. The two statements S_1 and S_2 share the same cache lines, so an execution of one will cause the other to be flushed from cache. The worst case behaviour occurs when the conditional expression exp causes a sequence such as $[S_1, S_2, S_1, S_2]$ to be executed, and regardless of the value of exp , this behaviour is assumed by Arnold’s approach.

The same problem is shared by the approach of Ferdinand et al. [82], which uses an alternative but broadly equivalent representation of abstract cache states. Ferdinand’s approach supports multi-way associative caches, and classifies instructions using a reduced form of Arnold’s groups (Figure 2.7(b)) in which “first miss” and “first hit” are considered to be “unclassified”. Mueller notes [183] that this “may lead to slightly more pessimism”.

All of these approaches have AI (section 2.1.2) in common: i.e. symbolic execution is used to determine timing properties. However, application of *implicit path enumeration* (IPET) is known to produce a tighter WCET estimate (section 2.1.2). It provides support for arbitrary constraints on execution, leading to a tighter WCET estimate. The technique can be applied in two ways: (1) at the execution path level only, using other techniques to model the behaviour of the CPU

```

00  for ( i = 0 ; i < 4 ; i ++ ) {
01      if ( exp ) {
02          S1 ;
...      ...
...    } else {
12      S2 ;
...      ...
...    }
... }

```

Figure 2.8: A code arrangement which causes Arnold’s approach to produce a pessimistic result. Assume a cache size of 10. The left-hand column shows the starting address of each piece of code. S1 and S2 conflict, because both map to cache lines beginning at 2.

Variable	Meaning
$f(x)_j^{hit}$	Lower bound on the number of times that a cache hit occurs during execution of line block j in basic block x .
$f(x)_j^{miss}$	Upper bound on the number of times that a cache miss occurs during execution of line block j in basic block x .
$\gamma(x)_j^{hit}$	Cost of a cache hit on line block j in basic block x (constant).
$\gamma(x)_j^{miss}$	Cost of a cache miss on line block j in basic block x (constant).

Table 2.4: Variables specified by the cache-aware integer linear programming model of Li et al. [160] (using symbols from Table 2.2 for consistency).

microarchitecture, and (2) at both the execution path level and the microarchitecture level.

In [160], Li et al. propose the second approach. They extend their IPET analysis approach (section 2.1.2) with new constraints that represent a cache model.

Li’s approach models an instruction cache as a series of “line blocks”, which are groups of cache lines that represent contiguous memory and are loaded from memory at the same time, which accounts for burst memory accesses. The line blocks used by each node of the execution graph are identified, and each line block is handled as a separate component of the total WCET. The variables of the constraint problem are as shown in Table 2.4. These are similar to the variables in Table 2.2, but the two cases of “guaranteed cache hit” and “possible cache miss” are separately accounted.

In addition to constraints specifying control flow, and constraints specified by the user to indicate loop bounds and other execution path properties, new constraints are added to represent the cases where cache hits can be guaranteed. These exist whenever a path exists between two line blocks that is not broken by a conflicting line block. As in other integer linear programming approaches [159, 207], all constraints are fed to a solver program which produces values for every $f(x)_j$, indicating the upper bound on each line block. The equation for the WCET changes from equation 2.1 (as in [159, 207]) to

$$\sum_x \sum_j (\gamma(x)_j^{hit} f(x)_j^{hit} + \gamma(x)_j^{miss} f(x)_j^{miss}) \quad (2.2)$$

Li’s approach includes a method of modelling multi-way associative caches based on a “cache state transition graph” that models all the possible changes of cache state during the execution of

the program. This allows LRU cache replacement strategies to be represented.

However, Li's approach has drawn criticism because the complexity of the integer linear programming problem is vastly increased by the number of cache-related constraints. In [280], Wilhelm reports that:

“the number of competing [line] blocks grows linearly with the size of the program...
Thus, the size of the [integer linear program] is exponential in the size of the program.”

Wilhelm's conclusion is that the approach as described cannot scale to analysis of large programs or more complex architectures.

2.3.3 Modelling Pipelines and Instruction Caches Together

Unfortunately, interactions between pipelines and instruction caches prevent the two components being considered in isolation. For example, a cache miss does not stall the entire pipeline: instructions that are already executing will finish as if a miss had never occurred. Analysis approaches must therefore account for the effects of both features.

In [162], Lim et al. propose the use of *worst case timing abstractions* (WCTA) to represent pipeline and cache states on the transitions between basic blocks. Like the abstract cache state proposed by Arnold [16], these represent information about worst-case pipeline and cache behaviour in a way that makes them amenable to merging. Whenever multiple paths lead to the same basic block, the abstract pipeline states of each path can be merged in a way that preserves the worst case properties embodied by the two paths, avoiding a need to enumerate all possible execution paths through the program. Lim also proposes composition operations on WCTAs for handling conditional and loop statements efficiently, and provides a proof of safety for an efficient loop-combining heuristic.

The WCTA approach relies on computation of “pipeline reservation tables” for each basic block. These contain information about the pipeline states at the entrance and exit of the basic block that lead to maximised execution time. The tables are computed as a first step, and Lim demonstrates that it is sufficient to compute a partial table with only δ elements at the head and tail. The tables are then combined using a graph of execution paths to find a WCET estimate.

A similar approach is used for cache modelling: the WCTA includes information about the cache state required by each basic block to avoid cache misses, and the cache state that remains when the basic block completes. This information is combined at the same time as the pipeline information, resulting in a worst-case miss count. However, this approach has been criticised [120] for not accounting for parallelism in the pipeline. By assuming hits during pipeline modelling, then subtracting the cost of cache misses a posteriori, Lim's model ignores the possibility that cache misses will be handled as instructions execute.

In [120, 119], Healy et al. extend Arnold's abstract cache state approach to include pipeline modelling. The pipeline modelling approach is equivalent to Lim's approach [119], but Healy's approach does not model the reservation tables. Instead, a minimal set of pipeline properties are captured (Table 2.5). These represent both structural hazards, when a pipeline unit cannot be assigned a new task because it is busy with a previous task, and data hazards, when a pipeline unit cannot begin execution because input data is not ready. This approach has the advantage of efficiently capturing the effects of instructions with very long execution times, such as floating point division. In [183], Mueller describes extensions to Healy's approach to support multi-way associative caches with a variety of replacement algorithms.

Record Name	Dimension	Purpose
Cycles from beginning	Num. Stages	Cycle when particular stage is initially occupied
Beginning Occupant	Num. Stages	Instruction that first occupies stage
Cycles from end	Num. Stages	When stage is last occupied
Ending Occupant	Num. Stages	Last instruction that occupies stage
Reg. first needed	Num. Registers	Cycle when value of register is first needed
Reg. last produced	Num. Registers	Cycle when value of register is last used as a destination

Table 2.5: Structural and data hazard model used by Healy for capturing pipeline information ([119], Figure 4 and Tables 2 and 3).

2.3.4 Modelling Multiple-Issue Pipelines

All of the approaches considered thus far have been limited to single-issue pipelines in which at most one instruction begins execution in each clock cycle. In order to exploit ILP, complex CPUs aim to issue multiple instructions in the same cycle (section 2.2.2).

In [163], Lim et al. propose a model for in-order superscalar pipelines. The model assumes that a CPU can begin up to k instructions simultaneously, and predicts which instructions will be executed in each clock cycle by modelling the behaviour of the in-order issue unit. In-order issue units must detect structural and data dependences between instructions so that no instruction begins execution before its input parameters are ready. Lim models this using an *instruction dependence graph* (IDG), a directed graph of instructions. This model predicts the maximum and minimum issue time for each instruction. As in Lim’s earlier publication [162], pipeline states can be merged in a way that preserves worst-case behaviour.

Lim’s model does not consider the effects of caching, but the authors state that previously proposed techniques for cache analysis, as proposed by Healy, Li and Ferdinand, can be applied.

For the task of complex CPU modelling, the major flaw in Lim’s work is a lack of out-of-order CPU modelling. In [158], Li extends Lim’s model to include an out-of-order CPU model, and uses both AI and integer linear programming in the manner described by Theiling [251]. This model also supports functional units that are themselves pipelined and able to process the different stages of multiple operations simultaneously. However, it is assumed that instructions are issued one at a time, even though out-of-order execution is possible. In [26], Barre extends this model further to support multiple issue pipelines: this increases the complexity of analysis.

2.3.5 Modelling Branch Prediction

Out-of-order operation effectively forces speculative execution, in which the outcome of a branch instruction is “guessed” before the instruction is actually executed. This increases the number of instructions available for re-ordering, providing more opportunities for ILP than available within a single basic block [268]. Speculation mechanisms may be static (for example, always assuming that a branch is taken), but dynamic branch prediction approaches are more common [233, 124, 89] because they are better able to account for typical program behaviour.

Some CPU architectures avoid the need for branch predictions by using *delayed branches*, in which every branch instruction is followed by “delay slot” instructions which are always executed

regardless of the outcome of the branch. However, this technique does not scale up to long pipelines because of the difficulty of filling each delay slot with a useful operation [195]. Superscalar CPU designs have avoided the use of delayed branches. Consequently, WCET analysis of modern CPUs must include branch prediction modelling.

Branch prediction is similar to caching in the sense that an incorrect prediction (a “miss”) will stall some parts of the pipeline, delaying execution. Additionally, an incorrect prediction may have side effects, perhaps altering the cache state because of fetching “wrong path” instructions. Therefore, branch prediction must be modelled. The difficulties are numerous: branch prediction units can be extremely complex [89]. (Perfect prediction is not possible, but more sophisticated heuristics can outperform simple heuristics such as a single taken/not taken bit on typical code.) A second difficulty is the need to properly account for aliasing effects, where two different branches are mapped to the same place in a branch prediction table. Additionally, while incorrect speculative execution has no effect on the meaning of machine code, it may nevertheless have an effect on the timing of that code. (For example, mispredicted execution might cause a variable to be flushed from the data cache.)

In [55], Colin and Puaut apply static model techniques previously used to model caches [119] to branch prediction. Their model is based on the Pentium CPU, and classifies each branch prediction using categories similar to those proposed by Arnold [16] (Figure 2.7(b)). Aliasing effects are considered, and the outcome of the model is an upper bound on the number of mispredictions in a program. A similar approach is taken by Li [158], where additional execution path constraints are used to model branch prediction and bound the total number of branch mispredictions in a block.

Considering the branch prediction modelling problem in isolation, Burguière and Rochange [42] propose the use of a *mixed branch predictor* as described by Patil and Emer [194] to address the aliasing problem. Branches are annotated with two bits that indicate one of three conditions: (1) use dynamic prediction, (2) assume taken, and (3) assume not taken. Patil and Emer proposed mixed prediction to reduce mispredictions on branches that defeated dynamic prediction heuristics: in [42], the technique is applied to prevent aliasing and thus simplify analysis of the branch predictor. An alternate approach to the problem is proposed by Bate and Reutemann in [29], who apply a program modification approach to avoid aliasing effects. This has the advantage of requiring no special hardware support, but it is still necessary to determine where aliasing effects may occur.

2.3.6 Practical Issues for CPU Modelling

Many features of complex CPUs can be modelled. The literature contains sample models for caches, multiple-issue pipelines, branch prediction and out-of-order execution. However, all of these features add complexity to models, forcing researchers to choose between increased pessimism and increased analysis time.

An example of this is the contrast between Li et al.’s approach to cache modelling [160] and the approach used by Healy [119]. In [183], Mueller contrasts the two approaches:

“the response time for [integer linear programming] changes from seconds for direct mapped caches up to hours just for 2-way associative caches while our method exhibits response times of less than one second (direct mapped) up to a few seconds... even for 8-way associative caches.”

The cache details modelled by low-level constraints in Li’s approach may reduce pessimism, but the extra complexity adds substantially to analysis time.

This problem has led to the adoption of mixed approaches in which AI is used to handle low-level microarchitectural details, and IPET-like approaches are used for execution path modelling only. Theiling and Ferdinand introduced this hybrid approach in [251], as an extension of their previous work [82] with support for more complex constraints, combining the advantages of AI for microarchitectural modelling (high speed analysis) with the advantages of IPET (reduced pessimism and support for complex behavioural constraints). In [280], Wilhelm notes that the combined approach has been adopted by the commercial WCET analysis tool `aiT`, and concludes that it is an effective way to determine good WCET estimates of programs on “quite complex processors”.

In [121], Heckmann et al. describe the creation of models for complex CPUs (ColdFire MCF 5307 and PowerPC 755) for `aiT`. This analysis application forces exact modelling of the CPUs, rather than a simplified experimental model. A cache modelling approach based on [251, 82] was used. Heckmann found that it was necessary to integrate all the steps of WCET analysis into a single operation in order to correctly account for interactions between pipelines and caches (branch prediction is not present in the CPUs considered). This increases (a) the complexity of the software, and (b) the execution time of the software in comparison to separate analysis. According to Heckmann’s conclusion [121]:

“The features that contribute to... average case performance, like caches, branch prediction, speculation or out-of-order execution, make it difficult to determine the worst-case performance. Modular, separate analyses of the behaviour of programs on these features cannot be done with sufficient precision due to the interdependences between different processor components.”

The implication of this result is that analysis models are not readily reusable. A tool to analyse one CPU will need to be partially redesigned in order to support another CPU. As the microarchitectural details of the CPU can all affect execution time, even using a different revision of the same CPU may force a redesign.

Heckmann also listed a number of desirable properties of CPUs to simplify modelling [121]:

- **Separate caches** should be used for instructions and data to eliminate possible interdependences between instruction and data accesses. This makes cache operation simpler to predict, because there is no need to handle situations where (for example) a data fetch could remove an arbitrary cached instruction.
- **Cache replacement strategies** should always lead to a known state. If cache contents are ever partially or wholly unknown, each subsequent access must leave part of the cache in a known state. The ColdFire 5307 is cited as an example of a CPU with a cache that does not have this property, because its *pseudo round robin* approach means that “a replacement in one cache set influences all other sets” [121]. In contrast, an LRU replacement policy leads towards a known state.
- **Branch prediction** should be static, because modelling dynamic branch prediction increases the complexity of static analysis.
- **Shortcuts** should be avoided. These are CPU features that detect special cases and handle them quickly, but the circumstances in which this is possible depend on dynamic conditions. These features are not present in experimental models, but may be introduced into commercial designs in order to speed up common cases. WCET analysis tools have to handle these cases correctly.

- **Out-of-order execution** should be “limited”, because all possible instruction interleavings must be considered statically during analysis to find the worst case.

Clearly, some of Heckmann’s proposals are at odds with high performance CPU design, where trends have led towards complex dynamic branch prediction schemes [89] and deeper pipelines [118]. And the modelling problem for some complex CPUs is even worse than Heckmann’s paper describes, because of the issue of timing anomalies.

2.3.7 Timing Anomalies

Timing anomalies are a property of some CPUs, first reported by Lundqvist and Stenström in [167]. The anomalies break an assumption that is present in all of the WCET estimation approaches examined so far, specifically that it will always be correct to assume the worst possibility. The effect is common in out-of-order CPUs, but as Wenzel [275] later demonstrated, in-order CPUs can also be affected.

Because evaluating every possible path through a program is an intractable problem, WCET estimation approaches [159, 119, 162, 121, 207] make two assumptions that permit multiple paths to a single basic block to be merged into a single “worst case CPU state”, specifically:

1. **Monotonicity**: that longer latency instructions (perhaps resulting from cache misses) can never reduce the overall execution time; and
2. **Composability**: that the CPU states resulting from multiple paths to a single basic block can always be meaningfully combined by taking the worst possible value of each property of each CPU state.

If timing anomalies can occur, these assumptions do not hold. Lundqvist and Stenström give the following examples of various types of timing anomaly [167]:

1. Cache hits can result in worst-case timing:

An out-of-order issue unit chooses an order for instruction execution based on a greedy algorithm that executes instructions as early as possible. A cache miss during a load operation may force a different order of execution to be taken, one that is faster than the order that would be taken if the load address was in cache because of better exploitation of parallelism within the CPU. (An example is given in [167], Figure 2.)

2. Cache misses may take longer to handle than expected:

The *critical path* through a series of operations is the path that defines the minimum execution time of those operations. Data dependences prevent any shorter path being used.

Lundqvist and Stenström give an example ([167], Figure 3) in which the critical path through a basic block is $[A, B, C]$, where A and C are load instructions and B is a multiply instruction. However, other instructions are also executing within the basic block, including multiply instruction D . If A is a cache hit, then B issues before D , resulting in best-case timing. But if A is a cache miss, then B issues after D , meaning that the execution time of D is added onto the time taken to process the critical path. This problem is similar to the *priority inversion* problem [43] in which a resource that is needed by a high priority task (in this case, the critical path $[A, B, C]$) is occupied by a lower priority task (in this case, multiply instruction D).

3. Impact on WCET may not be bounded:

Lundqvist and Stenström give an example ([167], Figure 4) of the *domino effect* in which a loop contains two independent instructions, A and B . A difference of a single clock cycle in the dispatch of instruction A on the first iteration is enough to change the order in which A and B execute, with the result that every iteration of the loop takes one more clock cycle in the example where A is delayed. In this example, the effect on the WCET value is proportional to the number of iterations. Again, the problem occurs because the out-of-order issue unit is using an earliest-first heuristic, and cannot consider the effects of each issue operation on the overall execution time.

In [275], Wenzel et al. generalised the results of Lundqvist and Stenström, identifying the sources of timing anomalies for all types of CPU. Wenzel characterised a timing anomaly as deviation from an expected behaviour, and gave a formal definition:

- In a program containing instructions I_1, I_2, \dots, I_n , the latency of each instruction I_i is t_{I_i} .
- The total execution time of the program is C .
- When the latency of the first instruction t_{I_1} is varied by Δt , the total execution time of the program is C' . The change in C is $\Delta C = C' - C$.
- A *timing anomaly* is present when either:
 - (a) $\Delta t > 0 \Rightarrow (\Delta C > \Delta t) \vee (\Delta C < 0)$, or
 - (b) $\Delta t < 0 \Rightarrow (\Delta C < \Delta t) \vee (\Delta C > 0)$

Less formally, a timing anomaly occurs when a change in latency causes an unexpected change in the program execution time. Wenzel characterised the causes of timing anomalies as *resource allocation decisions* using the following definition (from [275], definition 4.1):

“A resource allocation decision is defined to be possible in a hardware model M whenever an arbitrary instruction sequence S exists that may cause at least for one $u \in FU$ the instruction order relations $<^u$ and $<^{u'}$ to be different from each other $<^u \neq <^{u'}$ due to a latency variation by Δt of one single instruction in S .”

In this definition, FU is the set of all functional units in the CPU M . $<^u$ is the order relation of instructions of sequence S passing through functional unit u . $<^{u'}$ is similar to $<^u$ with the single difference that the latency of some instruction in some functional unit is increased by Δt . Using this definition, Wenzel proved that [275]:

“The possibility of a resource allocation decision... is a necessary, but not sufficient condition for timing anomalies to be present.”

Therefore, timing anomalies only occur in CPUs that make resource allocation decisions at runtime, selecting functional units and registers for instructions in response to both the instruction sequence and the pipeline state. The requirement to avoid dynamic resource allocations should be added to Heckmann’s list of desirable CPU properties for ease of analysis (section 2.3.6).

2.3.8 Accounting for Timing Anomalies

The timing anomaly problem puts an effective limit on the accuracy of WCET analysis approaches on CPUs that feature dynamic resource allocation. Unfortunately, this also puts a limit on the effective worst case performance of a CPU: average speed may be increased by a dynamic CPU feature such as an out-of-order issue unit, but it becomes increasingly difficult to guarantee a bound on the worst case timing. This motivates a search for ways to eliminate the timing anomaly problem.

In [167], Lundqvist and Stenström suggest that timing anomalies can be avoided by the introduction of instructions that force a known pipeline state, such as the PowerPC `sync` instruction. These would be introduced wherever variable latency instructions are present, and wherever a WCET timing analysis approach would need to merge pipeline states from two or more execution paths. However, the nature of timing anomalies also forces the cache state to be merged: the authors suggest this could be achieved by preloading lines that are needed by the next block but are not present in all possible cache states.

These adjustments make it possible to ignore the possibility of timing anomalies, and thus safely use an abstract interpretation approach for WCET estimation. Lundqvist and Stenström's experimental results suggest that this approach leads to up to 30% loss of performance on the PowerPC architecture.

Subsequent work by Li [158] provides a model that accounts for some types of timing anomaly, but simplifying assumptions are used in this model according to [72, 26, 217]. For example, Rochange suggests that Li's model may not be able to account for all types of timing anomaly [217]:

“Moreover, while this model seems to take into account most... inter-block effects, its ability to capture any possible long timing effect has not been proved... We feel that the model presented in [158] might miss this kind of effect.”

2.3.9 Trends Leading Away From Complex Models

Complex CPUs are clearly extremely difficult to model. Analysis tools must represent all parts of the CPU microarchitecture together in order to capture all possible interactions. Cache misses and branch mispredictions disrupt execution in ways that are difficult to model thanks to the effects of timing anomalies. And the general trend in CPU design is currently towards more complex CPUs, with larger cache hierarchies and more sophisticated branch prediction heuristics.

This has resulted in a paradigm shift amongst some hard real-time systems researchers, who advocate CPU designs and code generation paradigms that are specifically intended to be easy to analyse.

In [204], Puschner observes that the difficulties of WCET analysis are all related to branching. A program has $O(2^k)$ possible execution paths when k branches are executed on the longest path through the program. It is not possible to analyse all of these paths for a typical program, so AI is used to merge paths. Timing anomalies complicate this by defeating worst case merging assumptions (section 2.3.7), and the need to model every CPU component accurately complicates the analysis process.

Therefore, Puschner proposes the *single-path paradigm*, in which branches are omitted from programs in favour of predicated conditional execution. If conversion [3] is used to replace conditional statements with Boolean predicate values, which are used by the CPU to determine which subsequent instructions should execute. Loops are handled by unrolling to the maximum number

of possible iterations, and predicates are used to simulate the effect of leaving a loop early. When the predicate for an instruction is False, the instruction still executes as usual, but has no effect on the register file or memory. The paradigm is implemented by the SPEAR [61] embedded CPU, but any CPU with a conditional move instruction could be used instead.

Using this approach, the execution time of any program is constant. The WCET can be evaluated by simple measurement, with no need to model the CPU regardless of its complexity, as the program path is independent of the input. The disadvantage is that performance is reduced because all possible parts of the program are fetched and decoded. For example, both the `if` and the `else` parts of every conditional statement always run through the CPU, although at most one of these will actually execute.

An alternative simplifying approach that preserves both average and worst case performance is described in [9, 10] by Anantaraman. The *virtual simple architecture* (VISA) bounds the behaviour of an arbitrarily complex CPU to the characteristics of a simple CPU design that can be easily modelled by known WCET estimation techniques (specifically, those derived from Arnold and Healy’s work [16, 119, 183]). This is supported by adding two features to the complex CPU:

- A control line that switches off complex features that are known to be difficult to analyse, including dynamic resource allocation, dynamic branch prediction, out-of-order operation, and multiple issue. This causes the complex CPU to downgrade to a simple architecture: a mode change from “complex” to “safe”. The downgrade operation can be applied at any time for a small time penalty (the switching overhead).
- A watchdog timer that is used to monitor program execution on the complex CPU. Programs use this timer to detect situations when a checkpoint instruction has not been reached before an intermediate deadline. If a checkpoint is not reached in time, the CPU downgrades to the “safe” mode and continues.

In this arrangement, the WCET of a program can be computed by assuming that the simple mode is always used, and adding the maximum time between two checkpoints and the switching overhead. Checkpoints are spread throughout real-time tasks by the programmer or timing analyser.

The two modes allow the advantages of a high average performance CPU to be retained without also gaining the disadvantage of unpredictable operation. The real-time tasks have to be schedulable using the simple mode, so the principal advantage of the high performance mode is extra “slack” CPU time for non real-time tasks, which may even execute simultaneously with hard real-time tasks through an SMT arrangement. The slack time can also be used to save energy by scaling the CPU frequency. [9] reports that “a VISA-compliant complex pipeline consumes 43-61% less power than an explicitly-safe pipeline”.

A related approach is described by Rochange and Sainrat [217], in which a superscalar CPU is modified to ensure that the execution of each basic block is independent of previous basic blocks (excluding cache effects). This eliminates the problems caused by timing anomalies and simplifies WCET analysis, at the cost of also reducing the ILP that can be exploited. The authors note that this could be used to implement the simple mode in a VISA CPU, with improved performance and lower pessimism by comparison with the simple mode originally proposed by Anantaraman [9].

The VISA approach is promising, but it still makes use of caching. Cache analysis is possible, and techniques are now well known, e.g. [16, 183], but these approaches still introduce pessimism, and do not interact well with preemptive multitasking (section 2.3.12). Ideally, caching would not

be used at all. This is possible when small on-chip *scratchpad* memories (section 2.2.1) are used to store instructions and/or data.

Scratchpads are similar to caches in terms of size and speed, but do not include any auto-update logic or tag memory. They have been used to reduce program energy requirements [240, 139], but they are also useful as cache replacements for an RTS, because scratchpad access latency is very predictable. However, scratchpad space has to be explicitly allocated, often during or after compilation. Suhendra et al. [244], Puaut and Pais [202], and Wehmeyer and Marwedel [273] have all demonstrated effective algorithms for allocating scratchpad space to reduce a program's WCET. In [80], Falk et al. describe the closely related problem of allocating space within a locked instruction cache in order to reduce WCET.

Scratchpad and locked instruction cache allocations are driven by WCET analysis, which is used to detect worst case paths. Suhendra [244] and Puaut [202] use an IPET-like approach for this, while Falk [80] and Wehmeyer [273] make use of the `aIT` tool. However, it is not sufficient to perform just one WCET analysis, because each allocation may change the worst-case path through the program.

The problem is formalised by Falk, who introduces the notion of a *worst-case execution path* (WC path). This is the execution path that produces the WCET: its execution cost is greater than (or equal to) the cost of all other possible paths. This path is detected as a side effect of WCET analysis techniques.

Optimisations, such as code and data migration into scratchpad, can be performed to reduce the execution time of the path. This will reduce the WCET, but a new WC path may be created. Falk et al. classify WCET reduction approaches that do not search for a new WC path after each optimisation decision as *single-path analyses*, observing that they are not optimal. Falk, Suhendra and Puaut all consider this problem and handle it by reevaluating the WCET after each decision is made.

Suhendra's "greedy heuristic" for scratchpad allocation is shown in Figure 2.9. Although the heuristic that is used is quite simple (choosing basic blocks in order of their contribution to WCET) the algorithm is effective because it reevaluates the WC path. The algorithm assumes that the scratchpad is loaded before each task begins.

The Puaut and Pais algorithm for scratchpad allocation is shown in Figure 2.10. This algorithm closely resembles Suhendra's algorithm in terms of the process used to select basic blocks for inclusion in scratchpad. However, it can select more basic blocks per iteration and also considers partitioning programs into a number of regions, each with a separate scratchpad memory map.

Previous work [273, 244, 202, 80] has demonstrated that scratchpad allocation algorithms are effective. They allow a scratchpad to provide the same benefits as a cache without introducing any analysis difficulties. Scratchpads have been included in many recent CPUs [203], and it is likely that future RTSs will make good use of them.

2.3.10 Alternative Implementation Platforms

Another approach to simplify WCET analysis and reduce pessimism moves away from CPUs and software as implementation platforms. A *co-processor* is a hardware device attached to a memory bus. Co-processors operate under the control of the CPU, but carry out computations independently. Co-processors can be CPUs themselves, but application-specific hardware is also common. A co-processor is used to carry out a task independently of the main CPU, improving overall application

```

procedure Suhendra_Selection( $G$ ):
  ( $V, E$ ) =  $G$ ; to_be_placed =  $E$ 
   $Z$  = Calculate_WCET( $G$ )
   $r$  = Region(); change = True
  while ( Space_Remains()  $\wedge$  change ):
    change = False
    if ( |to_be_placed|  $\neq$  0 ):
      (1)   find  $x \in$  to_be_placed to maximise  $f(x)\gamma(x)$ 
      (2)   Allocate( $x$ )
            to_be_placed = to_be_placed - { $x$ }
            change = True
             $Z$  = Calculate_WCET( $G$ )
    end if
  end while
end procedure

```

Figure 2.9: Suhendra’s “greedy heuristic” for scratchpad allocation [244] evaluates the WCET of program $G = (V, E)$ on every iteration, finding the current WC path through G as it does so. It (1) selects the basic block $x \in E$ making the greatest contribution to the WCET by maximising $f(x)\gamma(x)$, and (2) allocates it to scratchpad, then repeats the WCET analysis. The loop exits when no further allocations are possible.

```

procedure Puaut_and_Pais_Selection( $G, N$ ):
  ( $V, E$ ) =  $G$ 
  to_be_placed =  $E$ 
   $Z'$  = Calculate_WCET( $G$ )
  list_bb = Select_Most_Beneficial_BB(to_be_placed,  $N$ )
  while ( |list_bb|  $\neq$  0 ):
    for  $x \in$  list_bb:
       $r$  = Get_Region( $x$ )
      Allocate( $x$ )
      to_be_placed = to_be_placed - { $x$ }
      list_reload_points =  $r$ .Get_Entrances()
      for  $r \in$  list_reload_points:
         $r$ .Schedule_Load( $x$ )
      end for
    end for
     $Z$  = Calculate_WCET( $G$ )
    if  $Z > Z'$ : return
    list_bb = Select_Most_Beneficial_BB(to_be_placed,  $N$ )
  end while
end procedure

```

Figure 2.10: Puaut and Pais instruction scratchpad allocation algorithm from [203, 202]. There are two major differences between this algorithm and Suhendra’s (Figure 2.9). Firstly, $N\%$ of all basic blocks are selected for allocation on each iteration by *Select_Most_Beneficial_BB*, which chooses basic blocks using the same heuristic as Suhendra (maximise $f(x)\gamma(x)$). In contrast, Suhendra’s algorithm only chooses one basic block per iteration. Secondly, the algorithm can consider partitioning: the program G can be divided into any number of non-overlapping regions, each having a separate scratchpad memory map.

performance (average or worst case). Co-processors can be designed to have easily-analysable timing properties without cost to performance. This is because there is no need to support a wide range of “average case” operations with a framework that can optimise for many different types of code.

This predictability makes co-processors highly desirable for real-time applications. For example, the Elphel 333 networked CCTV camera uses a co-processor for video encoding [84]. The encoding process is a hard real-time task implemented in hardware. The implementation is fast and predictable.

Various projects have aimed to integrate software and hardware tasks for real-time systems. These include the *hthreads RTOS* [1], which supports tasks implemented in both software and hardware. Scheduling services are provided for both types of task, and the RTOS manages communication between them. It is even possible to use the same language to write both software and hardware tasks: the *York Hardware Ada Compiler* (YHAC) [20, 272] is an example of a project to enable this. YHAC generates co-processor hardware designs from Ada code that operate with predictable timing. This has been used to create hybrid systems where real-time software is split between co-processors and simple, predictable CPUs [19].

2.3.11 Statistical Analysis Methods

It would appear that tight WCET analysis of present-day CPUs is effectively intractable because of the complexity introduced by dynamic features. Where analysis *is* possible, the analysis tools are expensive to develop and pessimistic. This motivates new approaches, as described in sections 2.3.9 and 2.3.10. Part of the difficulty is the need for an accurate model of the CPU.

However, analysis is possible without an explicitly programmed CPU model. In [34, 35], Bernat et al. propose *probabilistic WCET* (pWCET), an implicit modelling using statistical data gathered from CPU measurements as the program under analysis is executed. The data is used to build a probability function for the WCET value of the program. The approach differs from tree- and graph-based analysis approaches (as described in section 2.1.1) in the following ways:

- The CPU model is generated by measurement, rather than by a manual process.
- The WCET for each block of code is a probability function (based on measurements) rather than an exact value for the upper bound.
- The execution path graph is decomposed by combining probability functions in a manner appropriate to the relationship between nodes.
- A WCET estimate value is not computed. Rather, a probability function for the upper bound on the WCET is produced. A user determines an estimate of the WCET value based on the desired probability of a deadline miss, which can be set to a very low (but non-zero) value.

The decomposition process uses *copula theory* to combine probability functions. The mathematics used are explained in detail by [34]. The process is efficient ($O(n)$ in the number of distributions to be combined). The level of pessimism can be reduced by increasing either the number of samples or the probability of a deadline miss.

The advantage of this approach is that the system itself is the model. All CPU behaviour can be represented by sufficient sampling. The approach works for any CPU regardless of complexity, and is additionally able to model the behaviour of other parts of the system such as devices.

The disadvantages of the approach are that large numbers of measurements may be required, and the approach does not strictly adhere to the requirements of hard real-time systems design in that it always admits a very small possibility of a deadline miss. However, the authors argue that (a) the time costs of the approach are low in comparison to the costs of modelling a CPU, (b) pessimism is reduced, and (c) the approach allows probabilistic reasoning regarding the acceptable margin of safety, which is clearly superior to the common engineering practice of allowing an $x\%$ safety margin in which x is an arbitrary constant.

2.3.12 Multitasking

Unfortunately, there is an additional assumption inherent in all WCET analysis approaches, including proposals that reduce analysis difficulties such as VISA, single-path programming, and probabilistic modelling. This is the assumption that there is no inter-task *interference*: i.e. that no other tasks running as part of the system are able to affect the CPU state of the task being analysed.

A classical CPU can only execute one task at a time. Newer architectures such as multi-core CPUs and SMT break with this assumption, but these are disregarded here for the purposes of discussion. Using a classical CPU to execute multiple tasks requires some approach that multiplexes the computing resource between the tasks that need it.

For hard real-time systems, one suitable approach is *fixed priority preemptive scheduling* [21]. In this approach, each task T is assigned a priority P_T . The RTOS always assigns the CPU resource to the highest priority task that is ready to run. If the CPU is running task X , a task Y can preempt X if (and only if) $P_Y > P_X$. Once higher priority tasks have finished, or are waiting for some event, X resumes execution. Some of the context of X is restored by the RTOS to the state before preemption: for example, execution always resumes at the previous point. This means that the correctness of the computations carried out by X is unaffected by the preemption. But timing is affected, as the internal state of the cache, pipeline and branch prediction unit will not be restored. The RTOS has no effective way to save or restore the state of these CPU components, so the state is disrupted by the actions of the higher priority task. This disruption is interference.

In some arrangements, interference is not an issue:

- When the system only has one task, there is no interference.
- When the system has many tasks, but cooperative multitasking is used. This means that each task either (a) completes before the next task begins, as in a cyclic executive, or (b) reaches a known fixed point in the program where control may be handed over to another task. In this case, the effects of interference can be bounded because the CPU can only be reassigned to a different task at a few known points. A WCET analysis tool may assume that the cache is flushed at these points.

However, in general, preemptive scheduling can introduce interference at any point within a task. Only the highest priority task is immune to this effect. The effect also occurs when external interrupts are handled by software (these can be considered to be a form of high-priority task).

In a CPU with basic block timing invariance, preemption will not affect the lower priority task's execution time, provided that preemptions only occur at basic block boundaries. Interference has no effect, because there is no dependence on history. However, this cannot be arranged in many CPU architectures, so other methods are needed to model or prevent interference effects.

One solution to this problem is to account for the delay caused by preemption as the cache refills. In [155], Lee et al. describe a method for calculating the worst-case *preemption cost* associated with a task. However, even if the approach were to be extended to model preemption costs introduced by branch prediction and pipeline state, the problem of timing anomalies would still affect the accuracy of the results if the method were applied to complex CPUs.

Another solution is partitioning. This has been applied to caches [145, 182, 15], but could also be applied to branch prediction tables. In a partitioning approach, a cache is divided into subsections with a 1-1 mapping between subsections and tasks. Each task only affects the state of its own subsection, so interference is eliminated. This enables cache analysis approaches [16, 162] to be usefully applied to each task in a preemptive multitasking system.

An application of this is suggested by Mueller [182]: by combining Arnold's cache analysis approach with compiler-directed cache partitioning, inter-task interference can be prevented. However, the effective size of the cache available to each task is limited, and the total number of tasks is fixed by the total cache size. Therefore, the scalability of the approach is restricted.

A related technique is cache locking [200, 80], in which parts of a program are always present ("locked") in cache. This can be used to guarantee the performance of certain tasks, but again scalability is limited by the size of the cache. A similar idea is revisited by scratchpads [244, 202, 273].

In [15], Arnaud et al. consider a combination of cache locking and partitioning that attempts to improve the scalability of both approaches. Arnaud proposes that each task should be divided into regions at compile time, with each region requiring a different piece of code to be locked in cache. This idea is similar to *overlaying* [191], where a program is split into separate overlay regions which loaded into memory when execution reaches them. This enables one task to make better use of a small cache partition. The issue that is addressed is the problem of determining the most effective region boundaries: this is done by a greedy algorithm that partitions the graph of execution paths for each task based on execution trace data. The idea is extended for WCET reduction purposes in [201], where region boundaries are created automatically using WCET analysis information. A manual approach is described by Steinke et al. [240] where *copy points* are explicitly introduced into the program to create region boundaries.

2.3.13 Summary

All techniques for WCET estimation appear to be restricted by tradeoffs that (in general) force a choice between increased pessimism and decreased worst case performance:

- Simple approaches that assume independent execution of basic blocks (sections 2.1.1 and 2.1.2) are unable to account for the behaviour of complex CPUs. This limits the performance of programs analysed using such techniques: they must be executed on a simple CPU. This limits their applicability, even though IPET approaches are known to be highly effective when basic block timing invariance can be assumed [207].
- Complex CPU modelling approaches [121, 159, 119, 162, 275] are (one or more of): (1) costly to develop, (2) non-modular, (3) unable to scale to support large programs, (4) computationally expensive (with long analysis times), and (5) pessimistic. They cannot effectively account for timing anomalies. These issues limit the applicability of such techniques.

- Probabilistic modelling approaches [34] require exhaustive measurement. This limits the scalability of such techniques, despite the elegant probabilistic handling of pessimism. Repeated retesting with representative data sets is necessary to avoid bias.
- Approaches that simplify analysis problems (sections 2.3.9 and 2.3.10) show promise for implementation of hard real-time systems. However, scalability issues remain in co-processor approaches [20, 1] and the single-path execution paradigm [204, 61]. The VISA approach is promising [9], but caching is still being used, which introduces pessimism.

The problem that affects all of the WCET analysis approaches is the difficulty of analysing a CPU that dynamically adapts to programs, whether this adaptation is carried out in the cache, the pipeline, the branch predictor, or some other component. If dynamic resource allocation is not used, then:

- Timing anomalies are eliminated [275];
- Inter-task interference can be limited to cache effects, which may be resolved by partitioning [182] or eliminated by the use of scratchpads [240];
- Basic block execution times could become independent of execution history, enabling simple but elegant approaches to WCET analysis such as IPET (section 2.1.2).

However, a return to simple CPU architectures is not sufficient to solve these problems. There is a general trend towards greater application complexity as users expect more features within embedded systems and manufacturers reduce costs by assigning more tasks to a single CPU. Simple CPUs do not have the performance to support future applications.

This work aims to improve worst-case performance over simple CPUs (section 1.1) without reducing scalability. Consideration must now be given to approaches that improve performance but do not introduce the effects that make WCET analysis difficult, as many of the CPU technologies seen in section 2.2 have been found to do. These are reviewed in the following sections.

2.4 Improving Worst-Case Performance Using Software

Performance is the change in some measure of achievement per unit time. If the measure of achievement is the time taken to run a program, then improving the performance is equivalent to reducing the program execution time: either average (ACET) or worst-case (WCET). Performance can be improved by changing the CPU, or the program software, or both. Improving the software by itself may be advantageous:

- WCET analysis tools base their operation on high level source code and machine code, so optimisation techniques that are applied offline can be accounted for. WCET analysis approaches that include this feature have been demonstrated [183, 169, 162, 76].
- Software performance improvements can enable simpler CPUs to be used, because the software is more efficient. Simpler CPUs are known to be easier to analyse [121].
- Optimisations can be targeted at the WC path [80] in the software, as in [295].
- No application-specific hardware is needed, so the scalability problems that can apply to co-processors are not relevant.

In this section, methods for software performance improvement are classified and described. This begins with methods carried out by the programmer (section 2.4.1), followed by automatic methods (section 2.4.2). Section 2.4.3 discusses manual run-time specialisation and section 2.4.4 summarises the findings.

2.4.1 Manual Optimisation

The worst case performance of a program can be improved directly by the programmer using one or both of two types of approach: *algorithmics* and *optimisation*. These are examined in turn.

- **Algorithmics** is a formal part of Computer Science: the study of how a program specification can be most efficiently implemented. Algorithm selection can have a major effect on performance.

There is a calculus for characterising algorithmic efficiency which is often referred to as *big O notation*. Big O notation defines the upper bound on the amount of time (or memory space) that an algorithm will require in the worst case [58]. For example, consider a program that receives n inputs and requires $T = 2n^2 + 9n + 6$ machine instructions to process them. As n approaches ∞ , the quadratic term ($2n^2$) will dominate the linear and constant terms ($9n + 6$). Only the most dominant term is of interest in characterising algorithmic efficiency, so the big O notation for this program will be $O(n^2)$ (the coefficient 2 is also dominated by n^2). Clearly, the *time complexity* of the algorithms in a program is related to the WCET, but time complexity does not provide an exact relation between n and the execution time T . It describes an abstract relationship, independent of any computer architecture details.

The best known example of applied algorithmics is probably found in *sorting*. A sorting algorithm takes an input list of length n and re-orders it according to a comparison function, such as the numerical comparison $<$ (less than). Some sorting algorithms, such as bubble sort, operate with quadratic $O(n^2)$ time, as they carry out redundant comparisons, such as computing $x < y$ once it is known that $x < z$ and $z < y$. Even the well-known quicksort [58] algorithm degrades to $O(n^2)$ in some situations [184]. More efficient sorting algorithms such as heapsort [147] and introsort [184] always operate within $O(n \log n)$ time by avoiding redundant operations. For arbitrary large data sets, these algorithms are better choices than quicksort or bubble sort, although a simple $O(n^2)$ algorithm may be suitable when the data set is very small.

An optimal algorithm can yield excellent performance improvements over a simpler non-optimal implementation. For sorting, the change from $O(n^2)$ to $O(n \log n)$ can improve worst case performance by many orders of magnitude, depending on the value of n . This benefit is completely independent of the hardware used to execute the algorithm. An $O(n^2)$ algorithm should only be used in place of an $O(n \log n)$ algorithm in special cases where n is bounded and the $O(n^2)$ algorithm is faster for all possible values of n .

Implementation of an improved algorithm could be a costly process, requiring extra design, programming and testing steps. This is partly offset by standard code libraries such as the C++ *standard template library* (STL) [229], which includes generic implementations of widely-used algorithms and data structures.

However, libraries of this sort do not implement every algorithm. In some cases, the optimal algorithm for a task may be unknown. For example, no-one has yet (2008) published a

```

(a)                                     (b)
for ( r11 = 1 ;                          loop: l.sfeqi  r5,0x0
      r11 <= 32 ; r11 ++ )                l.bnf     exit
{                                          l.srli    r4,r4,0x1
    if ( r4 & 1 ) return r11 ;            l.addi    r11,r11,0x1
    r4 = r4 >> 1 ;                       l.sfleui  r11,0x20
}                                          l.bf     loop
return 0 ;                                l.andi    r5,r4,0x1
(c)                                       exit: ...
l.ff1    r11,r4

```

Figure 2.11: An example of some C code (a), which compiles to a loop in assembly code (b), but could be replaced by a single `l.ff1` instruction (c). This example uses ORBIS32 machine code (Appendix C, [151]).

method for decrypting messages encrypted with AES [187] that does not involve either knowing part of the encryption key or $O(2^k)$ attempts (where k is the key size in bits). It is not even known if a more efficient algorithm *could* exist. This example illustrates that performance improvement by algorithm selection is not always practical.

- **Optimisation** methods do not change the algorithm, but rather improve its implementation to reduce execution time. One approach is manual reimplementing of heavily used algorithms in a lower-level language (often assembly), where resource usage can be more tightly controlled than in the original language (such as C).

For example, consider the OpenRISC CPU [150] instruction `l.ff1` [151]. This searches a 32-bit word for the first non-zero bit, starting at the least-significant bit, and outputs the offset of this bit (starting at 1). This instruction is never generated by the `gcc` [94] compiler, because `gcc` is not able to detect the need for such a complex instruction. So `l.ff1` only appears within hand-written assembly, where it can speed some up processes such as interrupt dispatching. Figure 2.11 shows an example of some C-generated code that can be hand-optimised using `l.ff1`.

This type of optimisation has a long history. Compilers are not able to make optimal instruction selections in all cases, particularly in the case of CISC CPUs (section 2.2.2). For a CISC architecture, instruction selection is often carried out by pattern matching [2]. This strategy recognises sequences of low level operations that could be replaced by a shorter sequence of higher level operations. However, this approach is limited to the patterns specified by the compiler designer. As a result, some instructions are never generated. For example, the x86 `DAA` and `XLAT` instructions are never produced by `gcc` [96].

For a practical example of manual optimisation, consider the MMX [136] core found in modern x86-compatible processors. Generic C compilers (e.g. `gcc`) may not generate optimal MMX machine code. Therefore, programmers may manually translate C code to MMX assembly in order to maximise performance. The Pentium II architecture optimisation guide has some advice on this topic [133]. As an example, a *discrete cosine transform* (DCT) procedure written in C and MMX assembly can be found in the `mpg123` [126] audio player: the MMX version is used on x86 processors that support it, where it reduces the execution time.

```
while ( sz > 0 )
{
    *out++ = *in++ ;
    sz -- ;
}
```

Figure 2.12: A simple hotspot, which copies `sz` elements from `in` to `out`. The commands within the loop are executed `sz` times more often than the commands outside the loop.

This MMX DCT procedure is a *hotspot*: a frequently-used area of code [176]. DCT is a non-trivial operation, requiring nearly 780 instructions of MMX assembly code in [126]. Figure 2.12 shows a simpler example of a hotspot, which is a single basic block.

Hotspots are performance bottlenecks. When a programmer is interested in minimising ACET, it is normally sufficient to make use of a *profile* to find hotspots. The profile shows the number of times each part of the program is executed: the greatest numbers indicate hotspots, and these are targeted for optimisation. Profiles are produced by specially instrumented programs (e.g. programs built with `gcc -pg`), or by non-invasive monitors (such as a part of a virtual machine). The process is similar when the goal is to minimise WCET: the programmer targets the areas of the program that contribute the most to the total worst case execution time. These areas are easily located by both IPET and AI WCET analysis approaches, which provide a form of “worst case profile” through computation of the WC path.

The advantage of manually directed optimisation is the potential improvement of both ACET and WCET. All CPU capabilities are available to the programmer, so any instruction can be used, including ones that are not known by the compiler.

One disadvantage of this method is the cost of programmer time. Hand-written assembly is difficult to use and requires specialist engineering skills. A second cost is code portability. Hand-written code is inherently difficult to port to a different CPU architecture. This is why the DCT hotspot for `mpg123` is written in both C and MMX assembly: the C version is used on CPUs without MMX. For some CPU targets, the effort of writing the MMX DCT implementation has been wasted.

C compilers support hand-written code by the use of a directive such as `asm`. They also support direct linking against modules written entirely in assembly, including procedure calls between C and assembly, and can often produce the assembly code for any C function when a particular command-line switch is used (e.g. the `-S` option for `gcc`).

Algorithmic improvements and manual optimisation can improve worst case performance without impacting overall scalability or ease of timing analysis. However, both approaches are potentially expensive due to the cost of programmer time. Therefore, the methods themselves are inherently not scalable to a large set of tasks.

2.4.2 Automatic Optimisation

Manual optimisations can be very effective, but they are costly. Some of the same work can be done automatically by the compiler, with the result that both ACET and WCET are improved with only

a small addition to compile time. These automatic optimisations are not generally able to select better algorithms, but they are able to reduce the number of instructions required to accomplish a task in many cases.

It is important to distinguish between online and offline compilation. Examples of online compilers have already been given in section 2.2.2.7: generally, these produce code to be executed in the near future. In these cases, WCET analysis requires analysis of both the program being compiled and the compiler itself. The problem would appear to be intractably complex in the general case. Because of this, online compilers are not considered in this work.

In contrast, an offline compiler may run on a different computer to the output program, and an unlimited time may pass between compilation and execution. `gcc` [94] is an example. The output of such a compiler can be analysed to determine WCETs, and many existing WCET tools work in exactly this way [205, 280].

Early compilers maintained a close mapping between code statements and the CPU instructions that were generated to execute them. This meant naïve CPU resource usage. For example, the common C operation “`a ++ ;`” would always compile to a load operation, an increment operation, and a store instruction, even if the previous command had just updated `a`.

Modern offline compilers contain optimising procedures that remove redundant operations and try to keep commonly-used temporary values within fast CPU registers. Some algorithms for this are described in [2]. Well-known optimisations include common subexpression elimination, where identical components of calculations are detected and merged, and code motion, where invariant computations are moved outside of loops. The general aim of automatic optimisations is to reduce ACET, but WCET is often reduced as a side effect.

The efficiency of modern optimising compilers is so high that hand-coded assembly optimisations, such as those described in the previous section, are rarely necessary. CISC CPUs require specialised optimisers: [161] gives examples of optimisations that are specific to certain *digital signal processors* (DSPs). Even with these optimisations, CISC compilers generally only support a subset of the complete ISA. This is no longer as important as it once was, since modern implementations of CISC architectures such as x86 are themselves optimised for programs built by compilers.

In `gcc`, optimisations are controlled by command-line flags. A list for version 3.3 can be found in [93]. These include both CPU-independent optimisations like `-funroll-loops`, which replicates loop contents to expose more ILP within loops, and CPU-specific optimisations such as those activated by `-march=pentium-mmx`. There are also optimisation presets, such as `-O2`, which activate a particular set of optimisations. [140] summarises the effects of some `gcc` optimisations.

The advantage of software optimisation is an execution speed increase for a minimal extra development cost. The programmer does not need to understand how optimisations work in order to use them: the compiler only applies optimisations that are known to be *safe* in that the meaning of the source code is preserved. The high-level source code remains portable. However, there are some disadvantages:

- All areas of code are optimised equally. The compiler has no information about the behaviour of the program, and cannot detect hotspots for either average or worst case execution. Optimising everything is a compile-time waste of CPU cycles, but the cost of compiler time is dwarfed by the cost of developer time for manual optimisation of hotspots only.
- The close relationship between lines of source and machine instructions is lost. This is a

problem for developers, as programs need to be single stepped during debugging. It also affects WCET analysis tools, as noted in section 2.1.3, forcing special techniques to be adopted to track the effects of optimisations, such as compiler modifications that report optimisation transforms [183]. Alternatively, the compiler itself can explicitly aim to reduce WCET during operation, as Zhao et al. describe in [295].

- The optimiser is a program, and programs are not capable of leaps of insight of the sort required to reach an optimal solution in every case. This is why manual optimisation is still occasionally used even when an optimising compiler is available.

These disadvantages are only minor problems in most situations. Automatic optimisations are widely used by software developers for release code: they are normally turned off during development for debugging purposes. However, programmers sometimes assist an optimising compiler, providing more information to allow it to reach a better solution. Two methods for doing this are through *pragmas*, and through *feedback*. These are described below.

- A **pragma** is a hint from the programmer to a compiler, often compiler-specific. Pragmas direct code generation, optimisation, and the representation of data structures in memory. As pragmas are hints, rather than commands, compilers can ignore them (for example, if optimisations are turned off). Pragmas are included in program source code, sometimes as “out of band” data rather than actual code, and sometimes using keywords.

The C keywords `inline` and `register` are a form of optimisation pragma, respectively meaning *copy this function wherever it is used*, and *store this value in a register*. Carrying out these instructions is not mandatory for the compiler. If it ignores them, or cannot satisfy them for some reason (e.g. all registers are already allocated), the code will operate correctly, but perhaps not as efficiently as the programmer had intended.

Other pragmas are out of band (not part of conventional C source code), manifesting themselves as `#pragma` directives in the source code. The pragmas that `gcc` supports are listed in the `gcc` manual [95], which also notes that these exist “primarily in order to compile code originally written for other compilers”.

Some compilers support more advanced pragmas to direct complex types of optimisation. This is common in proprietary compilers such as Intel’s official C++ compiler. An Intel paper [134] describes pragmas to control data prefetching and loop vectorisation. Combined with performance analysis tools, these pragmas allow the performance of a program to be optimised for a particular target set of Intel processors.

Pragmas allow some performance tuning with minimal programmer effort: there is no need to write assembly code by hand. They can be used for either ACET or WCET reduction. The standard C keywords `inline` and `register` are fully portable, and the `#pragma` directive is often portable in the sense that interpretation of the directive is not usually required for correct compilation.

However, this is not universally true. Dependence on `#pragma` directives can result in broken code. In general, pragmas make code less portable, although they are an effective way to supply hints about the code to the compiler. They allow the optimisation effort to be focused where it is most needed and applied in an appropriate way. Their utility as a mechanism for fine-tuning performance optimisations is illustrated by the example of the Intel compiler [134].

$$\begin{pmatrix} 0 & 2 & 0 \\ 0 & 0 & 4 \\ 3 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 2y \\ 4z \\ 3x \end{pmatrix}$$

Figure 2.13: A sparse matrix multiplication.

- **Feedback-Directed Optimisations:** in the absence of information about how a program will execute, compiler optimisations are applied equally to all code. However, some C compilers (e.g. `gcc`) allow an execution profile from an earlier execution to be used as an input for the optimiser. Using this data, the C compiler can arrange for the most likely paths to take the least time.

Feedback-directed optimisations are useful for ACET reduction, although debugging may be more difficult as the machine code depends on the profile input, which may contain random noise (e.g. from timing data) [185]. However, they are not so useful for WCET reduction, because they are *single-path analyses* [80]. Each WCET reduction resulting from feedback-directed optimisation could introduce a new WC path, and there is currently no way to incorporate this information into a feedback-directed optimisation process at the program level.

However, if a compiler is integrated with a WCET analyser, feedback-directed optimisation can form a part of WCET reduction. Such a compiler is described by Zhao [295].

In summary, automatic optimisations are easy to apply, and can provide performance improvements within most programs. WCET analysis is slightly complicated by the need to account for the effects of each optimisation transform, but previous work demonstrates that this is possible. Feedback-directed optimisations can be applied to reduce ACETs, and (in certain circumstances) WCETs.

2.4.3 Manual Run-time Specialisation

Run-time specialisation gives a program the ability to optimise itself for a particular input data set. Consider *sparse matrix multiplication*. A naïve algorithm for general matrix multiplication has time complexity $O(n^3)$ (algorithmically faster methods do exist for general matrices [130]). Sparse matrices are a special case in which many values are zero. This enables optimisations for multiplication, since multiplication by zero always yields zero. Figure 2.13 illustrates a multiplication involving a sparse matrix.

If the operation illustrated in Figure 2.13 is compiled offline, the compiler can optimise away all but three multiplications: $2y$, $4z$ and $3x$. However, this depends on the matrix on the left being known at compile time. If that matrix is a program input, an inefficient general algorithm will have to be given to the offline compiler, because the locations of the zeroes (if any) are unknown at compile time.

Online code generation is a solution. Suppose that a dedicated matrix multiplication function is created dynamically as soon as the matrix is known, e.g. in an image transformation program. This new function would be specialised for one particular matrix, e.g. a 45 degree image rotation. More examples of cases where run-time specialisation is useful are described in detail by section 2 of [156].

Unfortunately, the usefulness of this *metaprogramming* [27] technique is limited by two problems. Firstly, the most common language for embedded systems programming (C) lacks support for it. Secondly, there is no clear way to determine the WCET of the online compiler and the code it produces.

The first problem has been addressed by projects such as Tick C [199], which adds a form of *lambda operation* to C. Lambda operations create new functions, often using information that is only available at runtime, and are commonly used in functional languages such as Haskell [114] where metaprogramming is a first-order language feature. Tick C allows almost any C code to be added to a program, and the new code can access variables from the original program. Related work such as DyC [105] and Tempo [172] also allow run-time specialisation of C code, but are restricted to converting variables to local constants (invariants) within code, rather than permitting compilation of arbitrary code.

However, the WCET analysis issue has not been addressed. In general, it would appear that the issues of online compilation (section 2.2.2.7) apply here, i.e. both the program and the compiler must be analysed together to obtain the overall WCET. Therefore, run-time specialisation currently appears to be unsuitable as a WCET reduction technique.

2.4.4 Summary

Section 2.4 has examined various approaches for improving software performance without alterations to hardware. Some of these are compatible with timing analysis, in that they can reduce WCET without introducing new analysis difficulties, with the proviso that code transformations for optimisation must be accounted for. In all of these cases, improvements are applied statically, before or during compilation. Dynamic optimisations such as run-time specialisation (section 2.4.3) are not amenable to analysis because of the need to account for the effects of online compilation.

Because software optimisation is a mature topic [2], it is known that improving software alone will never be sufficient to meet the requirements of this work. It will not be possible to take a simple CPU and optimise the software sufficiently to match the performance of a complex CPU. Therefore, it is necessary to look beyond software optimisations in an effort to satisfy the objectives of this work, towards approaches that introduce new hardware.

2.5 Application-Specific Hardware

In this section, implementation technologies for application-specific hardware are reviewed. This provides the necessary background for section 2.6 which describes known methods for WCET reduction using hardware. Two types of technology are examined: firstly, *hardware description languages* (HDLs) in section 2.5.1, and secondly, the kinds of physical *platform* currently available for application-specific hardware in section 2.5.2.

For the purposes of discussion, it is important to distinguish between physical hardware (e.g. silicon) and the logic built on top of it. Both can be referred to as “hardware”, but in this work, the physical hardware is referred to as the *platform*, and the logical functionality is referred to as *user logic*. The purpose of a HDL is to describe user logic. The purpose of a platform is to implement that logic.

2.5.1 Hardware Description Languages

A *hardware description language* (HDL) allows a logic circuit to be described by source code. HDLs are used as the “source code” for user logic generation, targeting an ASIC or FPGA hardware platform (section 2.5.2). HDLs are also used for simulation, hardware modelling, and the creation of test benches.

The best known HDLs are VHSIC (Very High Speed Integrated Circuits) Hardware Description Language (VHDL) [17] and Verilog [266]. They are industry standards, with widespread support from *electronic design automation* (EDA) tools, which act as “compilers” for HDL code, translating high level definitions to user logic descriptions for ASICs or FPGAs. The two languages have similar capabilities but very different syntax. Both are declarative languages that allow complex structures of logic gates and memory elements to be defined and connected together in an arbitrary fashion.

Both VHDL and Verilog have the advantage that logic is directly defined. HDL synthesis tools automatically optimise designs, carrying out logic minimisation and mapping certain structures into implementations that are known to be particularly efficient, but in general, the design is exactly what the designer has declared.

However, this is also a disadvantage, because the designer is forced to explicitly define control structures such as state machines. This makes it difficult to translate software code into hardware designs. To simplify design entry and improve integration with software languages, some HDLs implicitly generate the required state machines in an attempt to mimic a software language. Handel-C [45] is one such language. Programs are written in a C-like language that is synthesised directly to user logic. The Handel-C “compiler” allows sequential and behavioural code to be written, automatically generating logic as required to support this. YHAC [20, 272] is another example of a compiler that targets hardware. The major criticism of software-like HDLs is that they generally do not allow *any* low-level control of the hardware that is generated. This can lead to inefficient implementations, although this may be offset by the increased speed of design entry.

2.5.2 ASICs and FPGAs

This section describes the most common implementation technologies for application-specific hardware platforms. Firstly, an application-specific integrated circuit (ASIC) [288] is a custom-built IC, produced by the same silicon fabrication processes used to manufacture generic CPUs. An ASIC is customised for a particular use described by whatever user logic is built into it.

ASIC manufacturing processes have a high one-time setup cost, but a low per-unit cost [11]. A *photomask* must be produced to describe the layout of features on silicon. This is then used by a *fabrication plant* to produce (*fabricate*) the device. As a result of the high setup cost, ASICs are used for high volume embedded systems that will be sold to mass markets, such as televisions, DVD players and modems. Reuse of ASICs is common: specialist development corporations sell ASICs in bulk to downstream manufacturers, who use them as components in their embedded systems with the result that different embedded systems for a particular purpose often share the same internal architecture. For example, the Texas Instruments AR7 “router on chip” ASIC contains almost all of the parts required for a consumer-grade router [250]. This single ASIC has been used within many different embedded systems [164].

For an ASIC platform, user logic has to be mapped to a transistor-level implementation. The usual procedure makes use of *standard cells*: implementation logic that carries out primitive logic

functions. EDA tools are able to do this automatically, along with the subsequent steps of *placement* (allocating logic to physical locations) and *routing* (allocating connections between cells). These steps can also be carried out by hand at a much higher cost, but with potentially better results (e.g. higher maximum frequency or less area usage).

Because of the costs of producing new hardware, reuse of components is common. There is no limit on the size of a standard cell, and large cell designs with specific functions (e.g. CPUs) can be bought. These are known as *intellectual property* (IP) cores. They are sold as HDL code or as *hard macrocell* images for addition to ASIC photomasks. Another way to reduce the costs of an ASIC is to make use of a pre-defined layout. A *structured ASIC* [8], also known as a *mask programmed gate array* (MPGA), is an ASIC in which the arrangement of standard cells has already been defined. Since part of the fabrication mask has already been configured, the production of the final mask is cheaper [180].

Despite all of these cost-saving innovations, ASIC use is always limited by manufacturing costs. A “reconfigurable ASIC” would be useful, because it could be fabricated in bulk and then individually configured for large numbers of low-volume applications. Early forms of this type of device include *programmable logic devices* (PLDs): ICs that can be programmed with a specific *combinational function* [261]. The output of a combinational function depends only on the input [127]. Addition of two numbers is an example of a combinational function because the adder has no internal state; previous operations cannot have any effect on the result. PLDs are generally too simple to act as replacements for ASICs. The most complex forms of PLD (e.g. [227, 6, 291]) add storage elements but are still not well suited for implementation of arbitrary logic circuits because of their internal simplicity. They are best suited as containers for *glue logic* [92], linking more complex ICs together.

At present, the closest technology to a reconfigurable ASIC is a *field-programmable gate array* (FPGA). Like an MPGA, a typical FPGA contains a two-dimensional array for logic components. Unlike an MPGA, however, all the logic devices have already been built into silicon along with the hardware needed to store the configuration data that specifies the functions to be used. And unlike a PLD, an FPGA can contain arbitrary circuits (subject to a size limit). But the energy requirements of FPGAs are generally higher than those of MPGAs of similar density due to the needs of the reconfiguration hardware. An FPGA makes use of a very large amount of platform logic to allow reprogramming. Volatile memory is used to store configuration data, so an FPGA must be reprogrammed with new configuration data every time power is disconnected. This step is not required for a PLD and is not possible for an ASIC.

Most FPGAs are intended to be as generic as possible, so that any type of user logic can be implemented on them, although some also contain specialised hardware components to support particular functions, implemented in pre-built cells. These may include “block RAMs” (small random-access storage areas) and CPU cores [289]. Designers can integrate these components into user logic. The components could be built on the FPGA fabric, but higher speeds and logic densities are possible by direct implementation on silicon.

The reconfigurable logic in an FPGA is generally organised in a two-dimensional “chessboard” pattern, in which there are three types of cell (Figure 2.14). One type is a switch block (SB), which allows data to be routed from one part of the FPGA to another. Another is an *input/output block* (IOB), which provides an interface to a physical pin on the outside of the FPGA. The rest of the area is taken up by the third type of cell: the *configurable logic block* (CLB). This arrangement is very similar to the arrangement in an MPGA, and it is possible to migrate designs from some FPGAs to

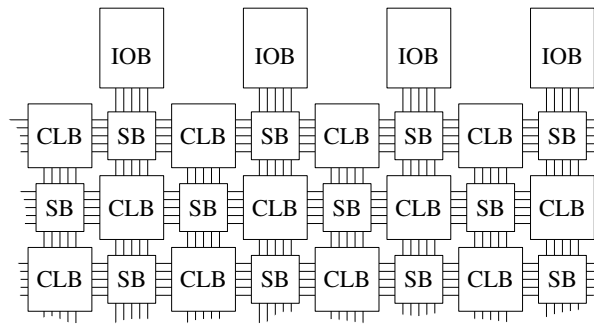


Figure 2.14: Reconfigurable hardware: the internal components of an island-style FPGA, composed of configuration logic blocks (CLBs), switch blocks (SBs), and I/O blocks (IOBs).

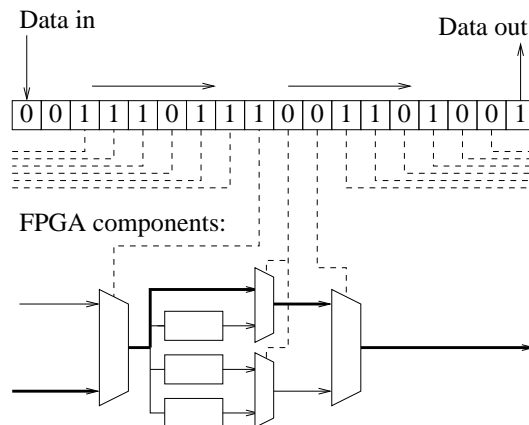


Figure 2.15: A shift register, as used to configure components within an FPGA. Configuration data is loaded into the shift register one bit at a time (left hand side). As it is loaded, the existing data moves to the right. When all data has been loaded, the FPGA components (below the register) are correctly configured. Shift registers can be arranged into chains of hundreds of thousands of bits to configure entire FPGAs.

some MPGAs in order to save energy and improve performance [210].

An FPGA is configured by a *bitstream* [283], a block of data that is designed to be loaded into the FPGA in a serial fashion. The bitstream contains low-level instructions to the switch blocks and LUTs. These instructions are distributed throughout the FPGA by configuration control hardware. Each configurable device has one or more *registers*, each storing one or more bits. These define the function of the device: programming a LUT or selecting a route through a switch block. The registers are organised as a large *shift register* (Figure 2.15), which minimises the number of logic gates required to manage configuration. Bitstream loading methods vary from FPGA to FPGA. Xilinx FPGAs support configuration by a parallel and serial interfaces, including via an industry-standard *joint test action group* (JTAG [193]) bus. [287] is a summary of supported methods for Spartan-3 FPGAs.

The process that converts a circuit description (written in some HDL) into a raw bitfile for FPGA programming is called *synthesis*. It is a software process that is normally carried out on a workstation, and it is similar in principle to ASIC mask design. But HDL code is mapped to CLBs instead of standard cells, and placement and routing are constrained by the resources that are

Feature	FPGA	MPGA
Volatile configuration?	Yes	No
Reconfigurable?	Yes	No
User Logic Speed	Slower	Faster
User Logic Density	Lower	Higher
Unit Cost	Higher	Lower
Custom Fabrication?	No, off shelf	Yes

Table 2.6: FPGA and MPGA comparison [180, 5].

available in the FPGA. Placement and routing are generally carried out by heuristic processes as described in [36], as both are NP-complete problems [249].

FPGA technology is far from being an ideal replacement for an ASIC. Table 2.6 contrasts FPGA and MPGA technology: it is clear that speed and logic density are lower for an FPGA. Energy consumption is also affected. According to [210]:

“In comparison with the Stratix FPGA, the HardCopy MPGA has an area reduction of up to 70%, an average of 50% performance improvement, and an average of 40% less power consumption.”

However, some FPGAs have capabilities beyond prototyping and ASIC replacement: *run-time re-configuration* (RTR). On an FPGA, the RTR process allows user logic to be replaced in one part of the FPGA without affecting the operation of other parts. This can enable *reconfigurable computing* [57], where an application includes descriptions of the user logic that it needs for operation, and is able to load them into a reconfigurable device. It can also permit user logic *virtualisation*, where reconfigurable hardware is used to implement a subset of a much larger area of virtual hardware through a paging process, analogous to virtual memory management [90, 129, 279]. Through virtualisation, reconfigurable computing could allow co-processors to be used in a scalable and flexible manner, as a reconfigurable platform could support any number of arbitrarily complex co-processors. RTR could also enable run-time specialisation [173, 37] for user logic, which is similar to run-time specialisation for software (section 2.4.3): a generic object is specialised to support a subset of functionality with greater efficiency.

But RTR support on Xilinx FPGA devices is limited in several ways. Firstly, placement is restricted. The *partially reconfigurable* (PR) logic can only be loaded into a reconfigurable module of the correct shape, with the correct size, with bus macros in the correct positions, and at the correct FPGA position [284]. This causes scalability problems. For example, the Karlsruhe RTR project (section 2.6.5) uses a series of s fixed-size slots as reconfigurable modules. Each piece of PR logic cannot be relocated, so each must be rebuilt for every slot. If p user logic components exist and full relocation is required, ps configurations must be built. Secondly, context data has to be explicitly handled. In a co-processor, the context is the state of all internal registers. Hardware context switching is difficult because dedicated save/restore data paths are needed, but context save and restore are required for full flexibility. Thirdly, other restrictions apply to the hardware designs, affecting clock and reset inputs ([284], appendix B).

2.5.3 Summary

This section has reviewed implementation technologies for application-specific hardware. Of the HDLs in common use, VHDL and Verilog are the most widely used. Both allow a high degree of control over the user logic that is generated for an ASIC or FPGA. Either would be ideal for the implementation of an architecture for WCET reduction.

Of the platforms currently in common use, FPGAs provide the best platform for experimenting with hardware designs. ASICs are primarily used for finished designs, due to high manufacturing costs. FPGAs also provide the possibility of RTR, which could allow co-processors to be virtualised, allowing WCET reduction processes to scale. However, RTR currently has many technical limitations, so alternative approaches may need to be found.

2.6 Improving Worst-Case Performance Using Hardware

Application-specific user logic cannot be used within all types of embedded real-time system, since adding an ASIC or FPGA is not always practical or cost-effective. However, when it can be used, opportunities for powerful and predictable performance improvements become available. These can be used to reduce both average and worst case execution times of tasks.

In this section, the WCET reduction capabilities of various application-specific implementation technologies are examined. Section 2.6.1 looks at co-processors, while sections 2.6.2 and 2.6.3 examine the properties of extensible CPUs (ASIPs). The link between both of these and classical co-design is explored in section 2.6.4. Section 2.6.5 examines the FPGA approach for run-time reconfiguration as an approach for WCET reduction. Several flaws are found, so other possibilities are examined in sections 2.6.6 and 2.6.8.

2.6.1 Co-processors

Co-processors operate under the control of the CPU but carry out computations independently. A typical co-processor usage scenario is as follows:

1. **Setup:** the CPU loads commands into a co-processor's registers, by writing to a memory address assigned to the co-processor.
2. **Execute:** the co-processor is activated. In simple scenarios, the CPU may *busy wait*, repeatedly polling the co-processor to check for a "finished" flag. In more complex scenarios, the CPU may proceed with another task until the co-processor signals completion via an interrupt.
3. **Read back:** the CPU reads results from the co-processor via the bus.

Co-processors can be built into the same integrated circuit as a CPU. In this case, they can be accessed by special instructions rather than memory operations. The floating-point processor integrated into recent members of the x86 family is an example of this. But co-processors lack many CPU-specific overheads, so migration of tasks to co-processors can lower both ACET and WCET. CPU overheads that are not present in a co-processor include:

- **Memory bottleneck:** CPUs must fetch instructions from memory, which limits the rate at which a program is executed. Co-processors do not fetch instructions.

- **Instruction rate bottleneck:** CPUs can only carry out a limited number of operations in parallel, so if the operation involves several calculations, they must be serialised. Co-processors can carry out operations in series or in parallel as required.
- **Data path overhead:** CPUs only support a limited range of data types. The overhead for working with a 32-bit number may be the same as the overhead for working with a single bit. Co-processor data paths can be tuned to match the data that they will carry.

In addition to the performance benefits, the removal of these overheads also benefits WCET analysis. There is no complex behaviour to account for: no cache, no dynamic resource allocation, and no possibility of timing anomalies. This is why co-processors have been proposed as part of the solution to the difficulties of timing analysis [19].

Co-processors are particularly efficient at handling *streaming data*. Streaming data is continuous information, such as compressed video or encrypted data. A *stream processor* applies the same set of operations to every part of an incoming data stream, often producing an output data stream in response. Because hardware is very efficient at applying a fixed sequence of operations in parallel, co-processors are often used as stream processors, e.g. for video compression [84]. In conventional code, stream processing often forms a hotspot, which can be offloaded to a co-processor for hardware acceleration in place of software optimisation.

However, co-processors are not a scalable technology unless combined with RTR at the platform level. There is no way to expand the task set of a co-processor since it cannot be reprogrammed arbitrarily. Co-processors are also poorly suited to implementing parts of a task, with other parts being implemented by software, since this requires communication between the CPU and co-processor to transfer state information. Additionally, co-processor design and implementation costs may be high, and HDL designs will tend to give a fixed tradeoff between performance and space usage, with a redesign being needed to change this tradeoff.

2.6.2 Application-Specific Instruction Processors (ASIPs)

Building a custom co-processor is an expensive undertaking. Additionally, co-processors do not integrate with software very well, because of the need to transfer parameter information from the CPU or memory. Software for generating *application specific instruction set processors* (ASIPs) provides a solution for both problems. ASIP generators produce CPU cores: the target platform is an FPGA or ASIC.

ASIPs such as Xtensa [103] are RISC CPUs with extensible instruction sets. They are “configurable CISC” devices in which the functions of custom instructions can be specified by a developer to match the needs of the application. These new instructions are implemented by additional hardware, but can make use of existing CPU elements such as registers, buses and functional units. Custom hardware components can be large and complex. For example, in an application described by [270], “a few additional instructions” need “18,000 additional gates”, and in another application, “image filtering and colour-space conversion” custom instructions need an “additional 64,100 gates”. For comparison, the unextended ASIP required only 25,000 gates [248].

ASIPs can support timing analysis. The current generation of ASIPs are simple processors based on RISC designs, with short pipelines. There are no dynamic resource allocation decisions, so all instructions have fixed execution times (excluding cache effects). This applies to both built-in instructions and custom instructions. The custom instructions can be seen as either CISC-style

```

(a)                                     (b)
operation Cust_Inst ( out int x ,      value1 = ( a + b + 4 ) & b ;
                          in int a , in int b ) value2 = Cust_Inst ( a , b ) ;
{
    assign x = ( a + b + 4 ) & b ;
}

```

Figure 2.16: Example of Tensilica Instruction Extension (TIE) code [53]. The TIE definition, (a), adds a new instruction to the Xtensa ASIP [103]. C source code for Xtensa can then make use of the instruction, using a macro that is generated for it by the Xtensa compiler. (b) shows two equivalent C computations: the first using conventional instructions only, the second using the new instruction.

complex instructions that replace large numbers of simple instructions, or as co-processors that are closely integrated with the CPU core. With direct access to CPU resources, overheads are reduced in two ways. Firstly, directly accessible register files eliminate the setup time otherwise required for copying parameters before operation. Secondly, because resources are shared between the CPU and the custom hardware, the total hardware size is reduced.

In general, co-processors are well suited to stream processing, and ASIPs are well suited to general purpose processing. Tight integration with the CPU allows them to provide both ACET and WCET reductions for any application, not just applications that operate on streams, although the degree of improvement may be smaller than a co-processor as general purpose CPU overheads are still present [24].

2.6.3 ASIP Techniques

Many CPU cores intended for implementation on FPGA hardware support some degree of customisation. Since these CPU cores are often supplied in HDL form, extensions can be synthesised into the CPU design when the FPGA bitstream is built (section 2.5.2).

The highest level of customisation is seen in ASIPs such as Xtensa [103], which provides support for application-specific instructions with a special custom instruction language (Figure 2.16), compiler and automatic ACET optimisation tool [247]. But customisation is also possible for other cores such as OpenRISC [150] and Microblaze [282]. OpenRISC allows extended instructions to be written in Verilog, subject to the restrictions imposed by the ALU (instructions must take one cycle, use at most two register inputs and produce at most one output). Microblaze allows some instructions to be disabled to save space.

The best example of such an instruction is division. In a Microblaze [286, 282] CPU, the division instruction `idiv` is optional. When it is turned off, the compiler uses a software procedure for division. This saves hardware space but increases execution time. Software division requires over 480 clock cycles on Microblaze (assuming all code is in cache and worst case parameters are given to the software division function). In contrast, hardware division with the `idiv` instruction always completes in 32 clock cycles [286]. This is a fifteen-fold WCET reduction, and the new WCET is independent of inputs or cache state. Migration to hardware division increases performance and predictability.

Given this, it would seem that the execution time of a WC path could be reduced by reimplementing

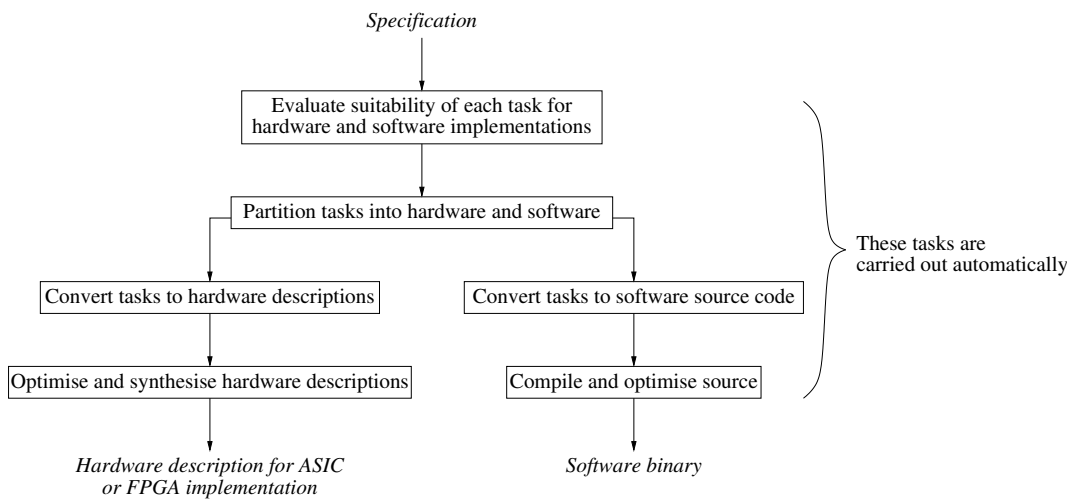


Figure 2.17: The idealised co-design process.

menting parts of it using TIE or some other custom instruction definition language. In principle, this would trade hardware space for a reduced WCET, and perhaps greater predictability. However, ASIPs introduce two scalability problems, preventing the number of custom instructions being increased indefinitely. Firstly, the set of custom instructions cannot be changed. The tight integration between the customised data path and the fixed parts of the CPU means that RTR is not practical on current FPGA technology due to the restrictions imposed by the platform [284]. Secondly, the size of the set of custom instructions is limited not only by space consumption but also by its effect on the maximum CPU speed. Adding new instructions to a data path means that more multiplexers are required and more routes are needed to move data to each new component. This always decreases the speed of the design, and since ASIP tools produce only HDL code, there is no way to limit this effect other than to use fewer custom instructions.

2.6.4 Classical Co-design

Co-processors and ASIPs provide useful frameworks for using application-specific hardware within an embedded system. However, both require some degree of manual involvement in the process of migrating software tasks to hardware. That process can be formalised as a *co-design* problem, because both hardware and software are being defined. Classical co-design is automatic optimisation, and the aim is to minimise system cost, development cost, energy cost, ACET or WCET.

Co-design has been studied fairly extensively. A goal of early co-design research was to create tools to generate optimised hardware and software simultaneously from a single specification [79, 107]. This included a search process called automatic *hardware/software partitioning* which attempted to assign parts of the specification to either a hardware implementation (as a co-processor on an ASIC platform) or a software implementation (on a conventional CPU). A common goal was ACET minimisation while remaining within a maximum size. Figure 2.17 shows the steps of the co-design process. Co-design has also been applied to real-time systems based on the aim of WCET reduction, e.g. [19].

Automatic co-design is limited in scalability due to the complexity of the hardware/software partitioning search process. It is not easy to predict the results of a partitioning decision, because

System	Approx. Scale
TOSCA [23, 226]	4kb of 68000 [181] code
Specsyn [98]	Programs for the Intel 8051 microcontroller
COSYMA [79, 122]	30-500 lines of C
Kalavade's system [143]	30 basic blocks
OCAPI [263]	2000 lines of C

Table 2.7: The scale of classical co-design systems.

the effects depend on the hardware that is already present, the communication resources that are available, and the size of the code being partitioned. A wide variety of search algorithms have been tried, including simulated annealing [79, 146], min-cut [262] and tabu search [73], but none are significantly better than the others. The search space is extremely large, and evaluating the costs and benefits of each possible partition is computationally expensive. The partitioning problem has been shown to be equivalent to an integer linear programming problem [209]. Since these problems have NP-hard complexity, the possibility of an exact solution seems remote for a large system. Indeed, the largest examples of automatic co-design are still small in relation to current embedded systems (Table 2.7).

Aside from this problem, co-design is limited by the general issues of using a single language to describe both software and hardware [106], and the scalability limitations of any approach that generates fixed co-processor hardware. This indicates that while co-processors or ASIPs could be used to reduce the WCET of a program, it will not be possible to allow a designer to make use of them without requiring additional work. In particular, a programmer will not be able to write a single high-level specification in a C-like language and then use an algorithm to determine which parts should be implemented by hardware and which parts should be implemented by software.

2.6.5 Using FPGA-based Run-time Reconfiguration

Run-time reconfiguration can allow physical hardware space to be shared by multiple co-processors (section 2.5.2) on FPGA platforms. This helps to solve the lack of scalability inherent in co-processor approaches. Each co-processor is likely to be highly specialised and non-modular, but allowing hardware resources to be shared between multiple co-processors permits one application to use as many co-processors as necessary. This is *hardware virtualisation* [90, 129, 279]. It is analogous to *virtual memory*, where a relatively small physical memory is extended to a relatively large virtual size by using a larger but slower memory to store the least frequently used pages of memory [222].

The following implementations of hardware virtualisation use the partial reconfiguration features of Xilinx FPGAs [281, 284, 285]. In each case, partial reconfiguration allows an embedded system to make use of many different co-processors, all sharing the same area of hardware.

- **Molen** [265, 149, 148] is a “polymorphic processor” with two parts - a conventional PowerPC CPU to run application software and an operating system, and an FPGA which can be partially reconfigured during application execution. The FPGA platform acts as a *virtual FPGA*, supporting dynamic loading of co-processors.

Molen's operations are directed before compilation by pragmas (section 2.4.2) which spec-

```

#pragma call_fpga op1
int f ( int a , int b )
{
    int c , i ;
    for ( i = c = 0 ; i < b ; i ++ )
        c += ( a << i ) + i ;
    return c >> b ;
}

```

Figure 2.18: C source code to be compiled for the Molen processor, from [265] figure 6. The pragma on the top line tells the compiler to translate the entire function f into HDL and replace calls to f with co-processor accesses.

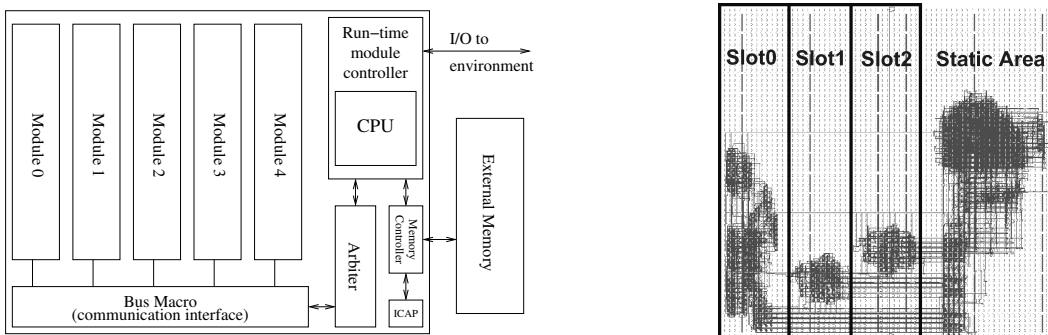


Figure 2.19: Pictures of the Karlsruhe dynamic reconfiguration project, both from [129]. Left: architecture diagram. The slots on the left side of the device can be loaded with co-processor logic by run-time reconfiguration. Right: a diagram of connections within the Karlsruhe FPGA.

ify software functions to be converted into co-processors. The translation from C source to HDL can be suboptimal [106], but as the hardware/software partitioning choices are entirely directed by the programmer, it is possible to optimise each function for hardware implementation rather than conventional compilation. Figure 2.18 is an example of Molen source code.

Molen uses a paging system, similar to a virtual memory manager, to manage the set of co-processors that are currently loaded. It can support an effectively unlimited number of co-processors by changing this set at runtime, enabling applications to supply their own co-processors with FPGA data included within the program binary. Thus, an application can use co-processors and achieve excellent performance improvements [265] without dedicating hardware to each task. The set of applications can be extended arbitrarily.

- The **Karlsruhe project** also uses dynamic reconfiguration [129, 260, 259]. Reconfigurable co-processors can be paged into several fixed-position “slots”, linked to fixed logic by a bus (Figure 2.19). This allows an arbitrary set of reconfigurable objects to be used simultaneously, but unfortunately each object must be rebuilt for every slot [259], and the slot sizes are fixed. These problems are due to the limitations of partial reconfiguration on current FPGAs [284].
- The **ETH Zurich project** [267] improves on this by using a single shared bus to link reconfigurable objects, permitting variable-size co-processors and relocation. The limitations of

partial reconfiguration are still present, but knowledge about the bitstream format is applied to allow objects to be transformed.

- The **Erlangen Slot Machine** [38] uses a second device for communication between co-processors, simplifying the difficult issue of efficiently handling communications between arbitrary co-processors.

It is clear that hardware virtualisation has inherent limitations. Some are due to the current technical limitations of FPGAs, while others stem from the inherent limitations of co-design approaches. For the purposes of WCET reduction, the issues are:

1. A co-processor model is forced (ASIP-style CPU integration and resource sharing is not possible), which would seem to limit the usefulness of RTR as a mechanism to reduce WCET, since co-processors are better suited to larger subtasks and automatic partitioning does not scale well;
2. Bitstreams have to be generated using slow processes such as place and route, and therefore cannot easily be applied as part of a larger process involving many iterative WC path optimisations;
3. Context data has to be explicitly managed [284];
4. Closed bitstream formats prevent new tools being written to generate bitstreams [91], so alternative approaches for bitstream generation cannot be attempted;
5. Bitstreams are not relocatable within an FPGA without a bitstream format-specific transformation tool [128], which limits the number of co-processors that can be loaded at a time;
6. Configurations can be large, creating storage problems.

These limitations do not prevent the use of such co-processors in hard real-time systems: indeed, Hübner [129, 260] has demonstrated a hard real-system using RTR. By allowing hardware to be virtualised, RTR avoids some of the scalability problems that affect fixed co-processors.

However, the first two limitations are severe problems in the specific case being considered here: WCET reduction of a general program. Consider limitation 1. Co-processors can reduce the WCET of software, but large sections of a program need to be migrated into a co-processor implementation in order to do this, because of the cost of communication between the co-processor and CPU. A single language cannot describe both hardware and software efficiently [106], and in general, co-processor techniques are limited by the co-design scalability problem described in section 2.6.4: finding the optimal boundary between hardware and software is a hard problem because searches do not scale well [73].

This problem is made even more difficult by limitation 2. The search for a good co-design solution involves many evaluations of possible hardware configurations, but each place and route operation is very slow. The specific case of WCET reduction worsens this problem further. A new WC path may be introduced by any partitioning decision, so it is not possible to evaluate partitioning decisions in isolation [80].

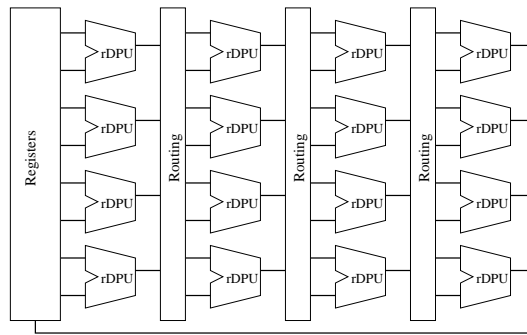


Figure 2.20: A simplified CGRA, containing three essential CGRA elements: rDPUs to carry out functions, registers to store temporary data, and routing resources for interconnection.

2.6.6 Coarse-Grained Reconfigurable Architectures

The various problems with FPGA-based RTR for WCET reduction force a search for alternatives. What is required is a specific form of RTR hardware that can support an ASIP-like arrangement. It must also allow configurations to be generated efficiently.

Some RTR platforms do not make use of FPGA features. Approaches using these have been demonstrated to be suitable for use within an ASIP-like CPU, such as [115, 264, 116]. All of these are CPU-integrated instances of a reconfigurable platform known as a *coarse-grained reconfigurable architecture* (CGRA) [110, 264], characterised as a connected grid of functional units and storage elements for implementation of data paths. This section examines CGRAs in an abstract sense, and section 2.6.7 examines their use within CPUs. Finally, section 2.6.8 looks at code generation approaches for such architectures.

CGRAs [110, 264] are superficially similar to FPGAs. The features of a typical CGRA are illustrated in Figure 2.20. CGRAs have been proposed both as an implementation technology for reconfigurable co-processors [101] and as reconfigurable CPU components [115, 264]. The physical differences between a CGRA and an FPGA are as follows:

- **Data path width** is greater than 1 bit. Within an FPGA, every logical connection can be any size, so data path sizes always match the requirements of user logic. In a CGRA, the fabric is optimised for a series of high level operations rather than arbitrary user logic, so connection sizes are restricted to the CGRA data path width. This simplifies placement and routing relative to FPGAs and improves the performance of operations that the CGRA supports.
- **Configurable logic blocks** implement operator-level functions [110] such as addition and subtraction, rather than low-level combinational logic as implemented by FPGA CLBs (section 2.5.2). In a CGRA, these CLB-like functional units are known by various terms. This work adopts the name *reconfigurable data path units* (rDPUs) after [113]. rDPUs are normally simple devices containing ALU-like computation features and storage in the form of registers, but there is no effective limit to the complexity of each rDPU.
- **Routing** may be constrained. Some CGRAs do not allow cyclic connections to be created, constraining user logic to tree-like designs [115], and some only allow routing in one dimension [224]. Others support near-arbitrary routing by FPGA-like hierarchies of routing

elements [178]. The use of constrained communication structures can dramatically simplify synthesis, placement and routing at the cost of flexibility.

CGRAs are logic circuits that can be implemented on any platform, including an FPGA or ASIC. On an FPGA, they form a new level of reconfigurable platform on top of the underlying hardware. The user logic for a CGRA takes the form of configuration instructions for the routing connections and rDPUs. Because functions are not bit-level, fewer instructions are required to specify each operation. CGRA configuration tools are simpler than FPGA tools, and the size of the configuration data is smaller.

A variety of CGRA-like designs have been investigated by researchers. In many, the CGRA fabric is a reconfigurable *semi-systolic array* - an array of interconnected data processing units that can efficiently process streaming data as it moves between adjacent rDPUs (section 2.6.1). The following list describes some CGRAs [110]:

- **KressArray** is a 2D array of rDPUs [113]. Verilog HDL code is generated from a specification, and this is then implemented on an ASIC [112]. Data path sizes and rDPU function sets are configurable, and an optimisation tool known as Xplorer is used for tuning [112]. The basic function set includes all the standard C operators. Each rDPU also acts as a router, transferring data between nearby rDPUs.

User logic is described in the ALE-X language [113], a C-like data flow language. ALE-X is a form of HDL. It is used for the same reason that other HDLs are used to describe general FPGA hardware: software languages are not adequate for the task [106]. Although KressArray configurations could be derived directly from software by automatic translation of C to ALE-X, the results would be suboptimal because standard C code does not provide information about opportunities for parallelism, loop vectorisation, and data path widths.

ALE-X specifications are mapped to a directed acyclic graph in which every node represents an operation carried out by a rDPU. This can be optimised by software techniques (section 2.4.2). The graph is then placed and routed on the target array by a search process based on simulated annealing. Figure 2.21 shows some results of this process. KressArray supports near-arbitrary routing, like an FPGA: each rDPU is connected directly to its neighbours, and longer routing resources are available, including a global communications bus.

The KressArray tools produce a configuration file that specifies the operations to be carried out by each rDPU, which can be a sequence rather than a single fixed operation. The small CGRA configuration size makes this possible, along with a feature for hardware context switching. Four different configurations and register sets (contexts) can exist within one KressArray [111].

Configuration sequences are used for multi-step operations such as multiplication and division are carried out in several steps. A *microprogram* is used as a controller. Microprograms consist of very low level machine code (*microcode*) with no intermediate mapping layer between microinstructions and hardware control lines. Microprogrammed commands can control both rDPU functions and routing dynamically [113].

KressArray is intended to be used as a co-processor in a larger system. It is flexible within a class of similar applications as it supports RTR for all or part of the configuration, but all KressArrays are tied to application-specific assumptions such as data path size and the quantity of routing resources. The authors note that “there is no general KressArray architecture

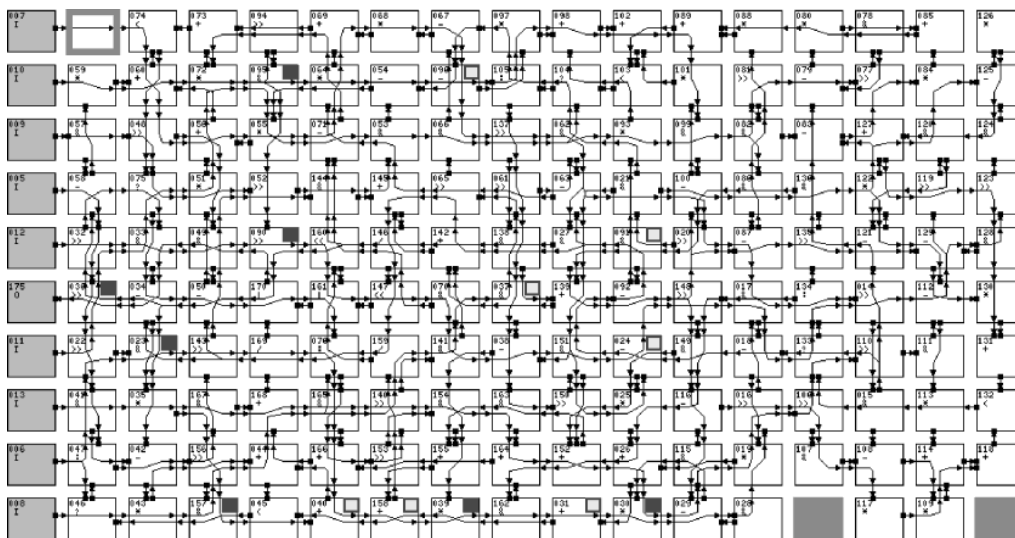


Figure 2.21: A filter circuit implemented on a 10 by 16 KressArray, from [112]. There are nine input ports and one output port, all on the left side of the array. The other boxes represent rDPUs. Each is assigned a function denoted by a C operator (e.g. + or >); more complex functions could be assigned to rDPUs to trade flexibility for a smaller circuit. Routing connections are shown by arrows linking rDPUs.

for any application” [112]. Additionally, the KressArray tools are reported to be resource intensive, preventing easy use as part of an iterative WC path optimisation process. Mapping the circuit in Figure 2.21 “took about 24 minutes on a Pentium-II 366MHz PC” [112].

- **REMARC** is similar to KressArray, but treats each rDPU as a small CPU [179], as an extension of the idea of configurations as part of a microprogram. Each rDPU, or “nano processor”, stores 32 RTR microinstructions, a small register file, a 16-bit ALU and input multiplexers. The microprogram sequence is global; so all the rDPUs operate in unison. The authors liken this to a 64-way VLIW processor [179]. There is no explicit support for context saving, but because of the nature of the control mechanism, it is possible to save and restore REMARC contexts by explicit addition of microinstructions for this purpose.

Like KressArray, REMARC is intended to be used as a co-processor. A main processor can load and execute REMARC microprograms by sending commands to REMARC. A custom HDL is used, rather than a C-based HDL such as KressArray’s ALE-X. The HDL is processed by a compiler to produce encoded microinstructions and mappings, which are then stored in C programs as embedded data (Figure 2.22). The C programs upload these instructions into REMARC as required, permitting many different uses for REMARC within the same application.

- **MATRIX** is a grid of 8-bit ALUs [178]. Like REMARC, it is controlled by a global sequencer and is intended to be used as a co-processor. Context save and restore is possible by adding save and restore operations to the microcode. The contribution of MATRIX is the addition of a PLD-like array at each rDPU. This provides a special purpose way to evaluate any combinational function as a predicate for other rDPU functions. The MATRIX interconnect provides near-arbitrary routing for data and predicates. As with REMARC and KressArray,

```

static unsigned int configuration_data [] = {
    0x01fe8245, 0x596f7520, 0x68617665, 0x20666f75,
    0x6e642074, 0x68652073, 0x65637265, 0x74206d65,
    0x73736167, 0x652e0a00, 0x0ee987fa, 0x01fe8245,
    .... } ;

```

Figure 2.22: Many CGRA approaches [179, 178, 116, 115] embed CGRA configurations as binary data within C source. Using this approach, the configuration data can be stored within program memory without any need for filesystem access, reducing loading time. The data is often treated as an opaque black box from within the C program, which does not contain functions to encode or decode configurations.

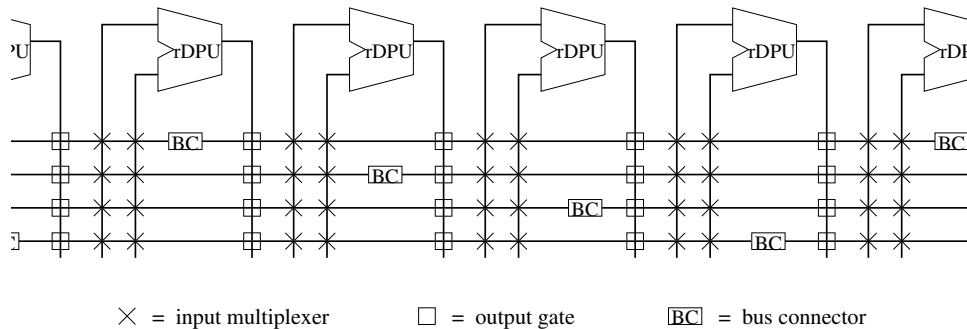


Figure 2.23: The Rapid interconnect [59] is a set of parallel segmented bus lines.

routing is run-time reconfigurable.

- **Rapid** is a linear (one-dimensional) CGRA [59, 68]. The CGRAs examined thus far have been two-dimensional: the most plentiful routing elements linked adjacent rDPUs, permitting easy implementation of user logic that required a fixed sequence of operations to be applied to a data stream. Rapid uses a set of parallel segmented bus lines for communication between rDPUs. No bus line runs the entire length of the array by default, but segments can be joined together to form longer connections (Figure 2.23). This arrangement simplifies mapping of user logic onto the array [59].

The Rapid control system is also of interest. MATRIX, REMARC and KressArray use global sequencers for control. Rapid uses both global and local sequencers. A global instruction from a RISC-like control CPU is decoded by LUTs at each rDPU, which can implement small state machines [68]. Rapid also makes a distinction between *hard and soft configurations* [59]. Each application for Rapid includes one hard configuration, which is static during the operation of that application, and one or more soft configurations which are dynamically controlled by the sequencer. Each soft configuration bit may change on every cycle.

Rapid acts as a co-processor for streaming data. Thanks to the flexible control system, it is able to cope with streaming operations involving complex operations including nested loops. Rapid configurations are sourced from an C-like HDL [110], which is compiled to Rapid configuration instructions by a process detailed in [67].

- **PADDI** is a hierarchical CGRA [52, 246], in which rDPUs are placed into clusters. A cross-

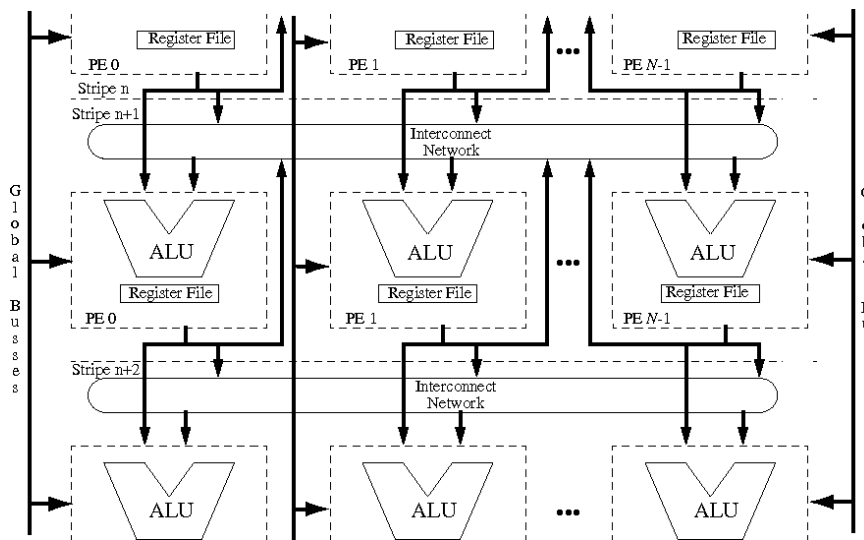


Figure 2.24: Diagram of Piperench CGRA, from [102].

bar interconnect is used to connect rDPUs within each cluster, and to allow connections between clusters. In PADDI-1 [52], each cluster contains eight rDPUs, and there is one cluster per IC. In PADDI-2 [246], each cluster contains four rDPUs and there are twelve clusters per IC. The intention is that multiple PADDI ICs will be used to prototype a DSP design.

PADDI has a custom toolset and uses an assembly language for circuit specification. The control scheme is similar to REMARC or MATRIX: a configuration memory at each rDPU decodes a program counter value into a large number of local control lines [52]. PADDI can support context save and restore through debugging connections, which allow external register access [246].

- **Piperench** is another type of linear CGRA [102, 224, 101]. In Piperench, physical rDPUs are organised as a cyclic linear pipeline, in which information is always moved in one direction. The pipeline is divided into stripes: Figure 2.24 illustrates stripes n , $n + 1$, and $n + 2$. Each stripe contains one or more rDPUs, each including an ALU and register file.

One of the most important contributions of Piperench is full support for virtualisation, where a small piece of hardware acts as a larger device with a performance penalty (section 2.5.2). A sequence of v pipelined operations can be virtualised onto p physical rDPUs even when $p < v$ by *pipelined reconfiguration* [102]. Using pipelined reconfiguration, Piperench can support complex streaming operations on a small physical array.

Piperench is intended to act as a co-processor, operating efficiently on streaming data. Configurations are designed using a C-like HDL known as *dataflow intermediate language* (DIL), which has the same purpose as ALE-X on KressArray, or the custom HDLs used by Rapid and REMARC.

The linear structure permits very fast placement and routing algorithms. An $O(n)$ algorithm is used [101]. In contrast to the search algorithms used for FPGA place and route [36], this algorithm never backtracks or reroutes [40]. The algorithm can result in poor utilisation of the array in some cases, and while it is acknowledged that “backtracking or local optimisation

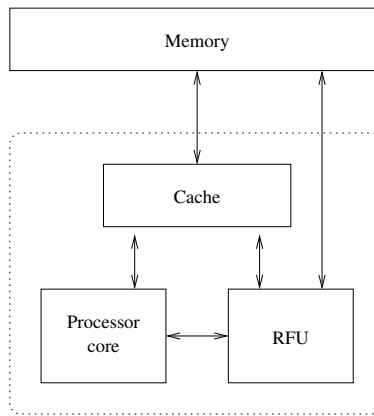


Figure 2.25: The Garp architecture

strategies might improve the result” [40], in practice the tradeoff between execution speed and array utilisation is believed by the designers to be correct [224].

- **Garp** [116] is an architecture that combines a MIPS CPU and a CGRA with the intention of offloading commonly used groups of MIPS instructions to the array. This is a contrast to the intended functions of the other CGRAs reviewed, which focused on efficient processing of data streams rather than code. The architecture is illustrated in Figure 2.25. The Garp CGRA has a data path width of two bits. A row of 16 Garp rDPUs form a complete ALU with support for all common arithmetic and logical operations. Garp is weakly coupled to the MIPS CPU by a co-processor bus, so data transfers must be explicit.

Garp has been designed with forward compatibility in mind, and the array is extensible. Configurations are embedded in C, as shown in Figure 2.22, and are activated by extended CPU instructions. However, they must be generated by hand, using an assembly-like code.

Garp has explicit hardware support for context switches, so no special configurations are required for saving and restoring context data. Available literature indicates that Garp has only been tested by simulation.

- **Mei’s CGRA** is similar to KressArray [175]. The contribution is the use of alternative algorithms for mapping user logic onto the array. Mei’s approach uses *iterative modulo scheduling* [87, 211] to carry out high level placement and routing as an initial guide for a conventional simulated annealing approach [174].

Modulo scheduling can partially vectorise inner loop code, providing a way to map some software code directly onto a CGRA without requiring a rewrite in a new language such as DIL or ALE-X. The operations in most software loops can be reorganised so that some occur in parallel. Greater parallelism is possible by overlapping iterations of the loop so that operations from multiple iterations are active at the same time within a *loop kernel*, which carries out the loop’s work. CGRA mapping tools usually attempt to expose this parallelism as a part of efficient mapping, but iterative modulo scheduling provides a way to do this using an established compilation technique.

All of the CGRAs reviewed in this section could be used to improve the performance of an embedded system. All are composed of rDPUs that include, at a minimum, functional units and

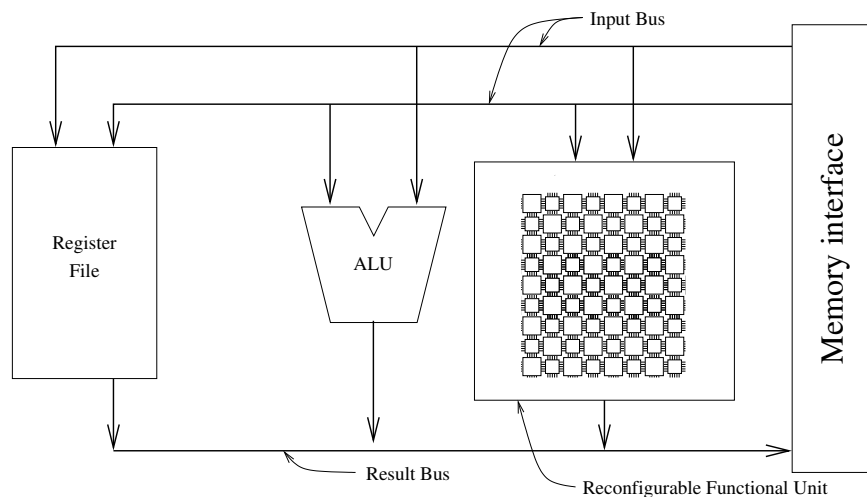


Figure 2.26: An idealised RFU architecture. The RFU is a CGRA-like component on the processor’s data path.

multiplexers, and may also include multicontext configuration memory and registers. The type of interconnect varies along with the computational power of each rDPU. In some cases, such as REMARC, rDPUs approach the complexity of CPUs. Context save and restore is generally possible, either by making use of CPU-like functionality to build an explicit save/restore microprogram, or by making use of a specialised hardware feature.

Most CGRAs use FPGA-like interconnects, allowing near-arbitrary routing between rDPUs. However, linear interconnects (as used in Rapid and Piperench) can work well. The Piperench tools operate efficiently thanks to the compiler-friendly design of the interconnect, with no requirement for heuristic searches.

But the CGRAs described above are all intended for use as co-processors. They require programming in custom languages [113, 101], or are specific to particular types of program [175]. What is actually required is a reconfigurable ASIP, rather than a reconfigurable co-processor. The next section explores previous work in this area.

2.6.7 CGRAs within a CPU Core

CGRA technology can be integrated into a CPU. To some extent, this has been demonstrated by Garp [116], where a CGRA is accessed through the CPU’s co-processor port. But in that arrangement, the CGRA is still a distinct co-processor, separated from the CPU data path and unable to directly access CPU registers. When the level of integration is tight enough to allow transparent interchange of information between the CGRA and other parts of the CPU, the CGRA is called a *reconfigurable functional unit* (RFU), as illustrated in Figure 2.26 [293].

The presence of an RFU allows a CPU to act as a run-time reconfigurable ASIP, or *reconfigurable instruction set processor* (RISP) [24]. RISPs combine the tight integration between functional units seen in an ASIP with the flexibility of RTR. They can be used for ASIP prototyping and can also provide ASIP functionality on a per-application basis. RFU approaches are well suited to speeding up common groups of CPU instructions by exploiting ILP, which is the method used for ACET reduction by many CPUs (section 2.2.2). Various projects have investigated RFU technol-

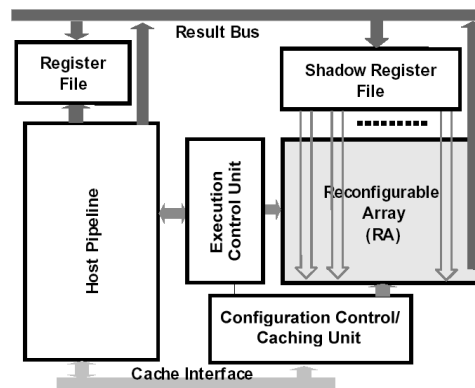


Figure 2.27: The Chimaera architecture (diagram from [115], page 2). Chimaera is tightly integrated into the processor, and has access to the current values in all the registers. The ECU and CCCU shown in the diagram are control units for the reconfigurable array.

ogy [24]. Of particular interest are:

- The **PRISC** RFU [213] is part of a MIPS CPU data path, with a similar architecture to Figure 2.26. Like other data path components, it has only two register inputs and one register output. PRISC is intended to bring performance improvements to any program.

The PRISC RFU computes a combinational function (section 2.5.2) of the two inputs using a network of small rDPUs. It is not capable of representing user logic requiring storage or memory access, but it can represent combinations of simple arithmetic and logical functions, so some sequences of ALU operations can be replaced with a single RFU operation. As PRISC contains no state, context save/restore is not necessary.

PRISC's RFU capabilities are limited in comparison to later types of RFU, but work on PRISC contributed tight integration between register file and CGRA, and the idea of storing configurations for RTR within the software then loading these configurations on demand. An interrupt is generated whenever software tries to access a configuration that is not already loaded into the RFU.

The PRISC tools include a compiler that extracts likely RFU configurations from program object code by combining conventional CPU instructions into RFU operations. This process is limited by the input bandwidth (two registers only). However, the PRISC compiler is able to extract constant values from the code and store these as part of the RFU.

- **Chimaera** [115] improves on PRISC by allowing the RFU read access to up to nine CPU registers, and write access to one CPU register per cycle. This is achieved using “shadowed” copies of the CPU registers, making data transfers between the RFU and other CPU components implicit and transparent without slowing down the CPU data path. The Chimaera architecture is shown in Figure 2.27. Like PRISC, there is no internal state other than the configuration, so context save/restore is unnecessary.

As with Garp, the Chimaera RFU has an extensible architecture for forward compatibility. Future CPUs could add more rows while retaining compatibility with first-generation Chimaera applications. The Chimera tools include a specialised C compiler [293] to handle the

translation of software to hardware, but partitioning decisions are made by hand. The configurations are stored within the program binary. An advance on Garp's approach is made through PRISC-style on-demand configuration. RFU configurations can also be compressed and a configuration cache is used to reduce average load times.

Chimaera has been implemented on silicon, and is reported to be effective at speeding up applications by up to 160 times [115]. However, the array does not contain storage elements and operates asynchronously (without clocking). This prevents access to memory, and limits storage to the single output register provided by the architecture, placing a restriction on the type of user logic that is possible. The time taken to produce a result depends on the length of the *critical path* through user logic: this is the slowest way for data to cross the RFU network. [293] describes various models for this latency, specified in terms of CPU clock cycles.

- **ReRisc** [264] improves on Chimaera through even tighter integration between the CPU pipeline and the RFU. ReRisc fits into a classic RISC-style five-stage pipeline [195] between memory access and register writeback.

Like Chimaera, ReRisc is only able to update a single CPU register at a time. But unlike Chimaera, the ReRisc RFU includes storage elements, permitting user logic to store temporary information within the CGRA. This means that operations can take several cycles, so the CGRA is more powerful. However, these registers are temporary: when the CGRA needs to store something that will be preserved across a context switch, it uses the CPU register file or memory directly.

This is possible because of tight integration between CPU elements and the RFU. Earlier RFUs act as ALU replacements: although the CPU instruction set is extended, the CPU itself is not redesigned to accommodate the RFU in an optimal fashion [116, 115]. ReRisc includes a redesigned CPU data path, optimised for RFU operations, and a control unit that allows access to sequences of RFU operations.

As with Chimaera, ReRisc uses a caching system to allow rapid reloading of recently used configuration data. Configurations are built into application binaries by ReRisc tools, as shown in Figure 2.22.

The trend in RFU design is towards higher integration between conventional CPU and RFU capabilities. Generally, an RFU acts as a reconfigurable ASIP, allowing common groups of conventional CPU instructions to be reduced to single RFU operations. This has been widely used to reduce ACETs [115, 116, 213], but has not been applied to reduce WCETs. However, an RFU would seem to provide an ideal way to reduce the WCET of a general program, due to:

1. the support for acceleration of any code within a larger program, rather than specific types of subroutine such as stream processors;
2. the support for RTR within a CPU; and
3. the highly predictable timing of a hardware component that does not use any form of dynamic optimisation.

An additional advantage of an RFU-based approach versus an ASIP approach is that all possible configurations have the same effect on the CPU maximum frequency. Because the functional units are placed on the data path at design time then configured by software, an ASIC or FPGA designer

can optimise the layout to maximise the CPU speed without knowing exactly which configurations will actually be needed. In contrast, the maximum frequency of an ASIP design will depend on the set of configurations being used, and optimisations will have to be repeated if that set changes.

An RFU-based approach can scale to support any number of tasks and any level of complexity within each task because the configuration memory can be reloaded at any time. The process is similar to the management of a scratchpad or locked cache. In [240], Steinke et al. add explicit “copy points” to a program to enable the limited space within a scratchpad to be used by an unlimited number of program elements. Although their intention is reduction of ACET and energy consumption, the same idea could be applied to reduce WCET. In [201], a program is divided into partitions automatically using WCET information, with copy points on each partition boundary. In [80], Falk et al. apply similar ideas using a locked cache.

2.6.8 Compiling Code for an RFU

Both CGRAs and RFUs require code generators. CGRAs require a stream programming language such as ALE-X since the language must describe a pipeline of parallel operations. However, since RFUs are replacements for parts of a conventional CPU, code can be generated from a conventional language: C source, or even machine code. Previous work exists in this area, since adding a programmable function accelerator to a CPU is not a new idea. It is very similar to extending a CPU instruction set by *user microprogramming*.

Some CPUs make use of a *microprogram* to carry out internal operations such as fetching and executing instructions. The microprogram acts as an interpreter for machine code and a controller for the CPU functions. In classical microprogrammed CPUs such as the Motorola 68000, the microprogram is stored in a ROM within the CPU [257]. It is divided into *microinstructions*, which are analogous to instructions expressed as machine code. But there is no intermediate ISA between the *microcode* and the CPU functional units. Each bit of each microinstruction controls a function directly.

Microprograms are not used within all CPUs because the same functions can be carried out by logic alone. A sufficiently complex state machine can replace a microprogram, and may well be faster. For example, Amdahl constructed a System/370-compatible CPU using a “hard wired” control unit which was able to execute code more quickly than IBM’s original microprogrammed design [198]. However, using a microprogram can reduce costs [195] and allows late design changes to be accommodated. When complex sequencing is required for certain machine instructions, using a software-like approach makes sense. Microprograms continue to be used within recent CPUs such as the Pentium 4 [135].

Because microprograms represent another layer of software, there is a possibility of changing a CPU’s microprogram to better accommodate a particular program. Such *user microprogramming* approaches were examined by researchers in the 1970s [252, 218, 219, 71, 30], primarily seeking ways to reduce ACETs. User microprogramming can improve performance in three ways: (1) by reducing the effects of the memory bottleneck, which does not apply to code fetched from microprogram RAM within the CPU; (2) by eliminating the overhead of fetching and decoding machine code; and (3) by allowing some operations to be executed in parallel (if the CPU provides sufficient resources for this). In short, custom instructions are created by extending the existing microprogram. This is very similar to the functionality offered by RFUs such as Chimaera and ReRisc (section 2.6.7), as an existing ISA is extended with application-specific instructions to reduce ACET.

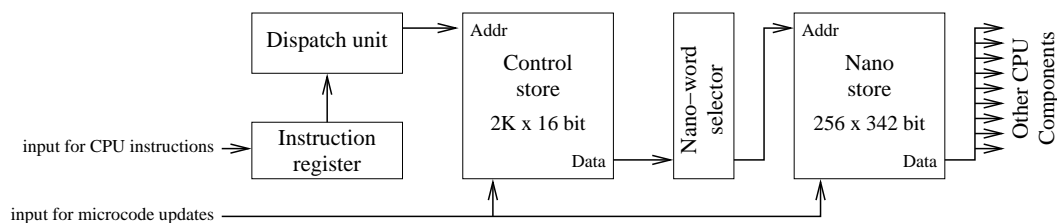


Figure 2.28: The two-level control indirection in the Nanodata QM-1 saves space by allowing microprograms to make use of a “palette” of 256 nano instructions. The microprogram store and the nano instruction store can both be reprogrammed during execution.

However, microprogramming can also reduce WCET, since dynamic CPU optimisations can be omitted.

Few current commercial CPUs support user microprogramming. To allow user microprogramming: (1) the microprogram memory must be RAM, (2) a code generator must exist for microprograms, and (3) an update port must be provided to allow the RAM to be changed by programs. Providing these features would raise the cost of a CPU built using 1970s technology, and although modern CPUs such as the Intel Core 2 Duo implement microprogram memories using RAM, the purpose of this is only to reduce fabrication costs and allow the manufacturer to resolve implementation bugs after sales [177]. Users cannot provide their own microprograms since there is no publicly available code generator and updates must be digitally signed.

However, in addition to the possibility of program ACET reduction, user microprogramming can allow a CPU to execute multiple ISAs. Historically, these possibilities have been a good motivation for the construction of specialist CPUs with a *writable control store* (WCS) feature, permitting user microprogramming. Notable projects that made use of a WCS feature include:

- The **Nanodata QM-1** CPU [218, 88, 62] is part of a 1972 minicomputer design for research into user microprogramming [218]. The CPU design includes two writable microcode memories (“control store” and “nano store”), which act as a two-level indirection table (Figure 2.28). The nano store is a *horizontal microcode* RAM, with a data bus that is much wider than the address bus, similar to the reprogrammable control store of each rDPU in some CGRAs such as PADDI [52]. The two level memory saves chip area by allowing “nano instructions” to be reused by microinstructions: the same idea was later used to implement the Motorola 68000 CPU [257], which used both a *vertical* (narrow instruction) and a *horizontal* (wide instruction) control store (in ROM) to reduce the transistor count of the microprogram memory.

Research using the QM-1 demonstrates the benefits of user microprogramming. Features necessary for scalability have been demonstrated such as control store virtualisation [219] which can allow many different applications to make use of their own set of customised instructions. OS services have also been implemented in microcode, like the TASK hypervisor described in [88], and these services can multiplex control store resources between many applications. Complete emulators for other CPU ISAs were also demonstrated [88], including the contemporary PDP-11 CPU [62]. The virtual machines that carry out this emulation are implemented using QM-1 microprograms, so QM-1 CPU resources are not wasted in executing a software emulation layer (e.g. a bytecode interpreter). The architecture’s flexibility also

allows microprograms to be debugged using software.

- The **Perkin-Elmer 3220 CPU** provides WCS access, and this could be made available to Unix programs through a driver [219]. In this arrangement, any Unix application on the machine can make use of user microprogramming without needing to consider other applications and multitasking. Microprogram store space is a form of virtual memory.

However, it is possible to crash a CPU by writing incorrect data to the microprogram device, and a malicious user could use the microprogram device to gain superuser privileges. In general, an OS can enforce the boundaries between applications, but it cannot detect illegal microcode. These problems apply to any multiuser system that provides user microprogramming services, suggesting that it may be unwise to allow unprivileged users to change the microprogram on such systems.

- **Automatic microprogramming** was demonstrated in [71] using a MAC16 computer. Groups of commonly-used instructions can be merged into complex custom instructions by a microprogram generator: this process is called *microprogram compaction*. The microprogram generator described in [71] carries out this task in response to a profile of system activity: like a JIT compiler, it optimises the current hotspots to reduce ACET.

Making use of a WCS is just one of the challenges faced by these projects. Another challenge is the need to generate microprograms algorithmically. This is essential if the approach is to scale to programs of any size. To some extent, well-known compilation techniques (e.g. [2]) can be applied to generate microcode: after all, microcode is just machine code for a very low level ISA. However, some of the differences between microcode and machine code introduce new issues that must be handled effectively. These issues are as follows:

- Microinstructions can specify **explicitly parallel** operations. Each may include one or more *microoperations* to be executed in parallel. A microprogram generator needs to make use of this parallelism to reduce execution time (through ILP) and reduce the total space needed to store the microprogram. This problem does not need to be addressed by RISC or CISC machine code generators, because they produce only sequential code.

Intuitively, it might seem as if any sequential code generator could be adapted to produce parallelised instructions by simply packing as many operations as possible into each microinstruction while preserving functional equivalence. However, this *microcode compaction* process is not as simple as it sounds, even in an idealised architecture in which all functional units are equivalent to each other.

The compaction problem has been examined fairly widely [157]. In [152], Landskov et al. present a survey of *local compaction* algorithms which compact microoperations within a single basic block. This is a special case of the general compaction problem which disregards the additional ILP that can be exploited if operations can be moved across basic block boundaries (speculative execution). This reduces the ILP that can be exploited [268, 186], but it also simplifies the work needed to handle control flow splits (branches) and control flow joins (branch targets). Within a basic block, there is only one control flow path, so assuring functional equivalence is a simpler problem.

A *global compaction* algorithm such as [253] permits operations to be moved across basic block boundaries, but this vastly increases the complexity of the search space. Compaction

across split and join points adds new paths along which functional equivalence must be assured. In [85], Fisher identifies several problems with naïve approaches to global compaction, characterised by the general problem that locally suboptimal compaction decisions may need to be made in order to reach an eventual global optimum. This is exactly the type of problem that is difficult to solve algorithmically, and has forced the use of heuristic searches such as simulated annealing in related problems such as place and route [36]. Searches of this type would slow down a general WCET reduction search process.

- Microprograms are **implementation specific**. Because microinstructions control hardware directly with no abstraction layer, microcode is highly specific to a particular CPU model. Machine code is guaranteed to have the same meaning on every implementation of a particular ISA, but there is no such guarantee for the underlying microcode.

The problem motivated IBM to refuse to support user modifications to the control store of the System/360 [7]. Such modifications could lead to improved performance, but they would also result in programs that would only work on a single type of S/360 CPU. All S/360 CPUs share a common ISA, but use different microcode to implement it [258], so third-party microcode could never be portable. While modifications are possible [30], a lack of official support makes them non-trivial.

- Microoperations may have **side effects**, which increases the difficulty of automatically matching program requirements to hardware capabilities. There may be several ways to implement a particular operation, but each may have side effects or constraints on its operation. These characteristics are beneficial when the CPU is executing the ISA it was designed for: side effects they might allow fewer microoperations to be used to implement a commonly-used instruction. But just like some CISC-style instructions (section 2.2.2), the added complexity makes it difficult for an automatic code generator to make use of the instructions.

Since these problems were first identified, some solutions have been proposed. However, before these are examined, it is worth looking at the theoretical basis for user microprogramming as an ACET reduction technology. The same properties that make user microprogramming useful for ACET reduction make it useful for WCET reduction.

Consider the software running on a computer as a multi-level structure delimited by interfaces, as shown in Figure 2.29. The top level is composed of applications: lower levels include software libraries, the OS, the microprogram, and finally the logic gates within the CPU. The example that is illustrated in Figure 2.29 shows an operation that spans every level of the architecture - an application invokes a library function, which invokes a system call, which invokes many instructions, each of which invokes certain hardware operations at the lowest level. Of course, most operations in a program will not invoke library functions or system calls: these skip directly to the ISA level.

But every operation does cross at least one interface between levels, and incurs a time penalty by doing so [242]. In [238], Stankovic modelled this process by introducing the notion of a prologue and epilogue. The prologue sets up the operation and decodes the action required: an example would be the system call handler in an OS, which typically begins with a state-saving step, and then carries out a table lookup to find the correct function for the requested call. The epilogue reverses this process if necessary. Both prologue and epilogue can introduce a time penalty, and this is always independent of the work done: it is the overhead of the level transition. Table 2.8 gives an example of penalties incurred by level transitions, re-using the levels from Figure 2.29.

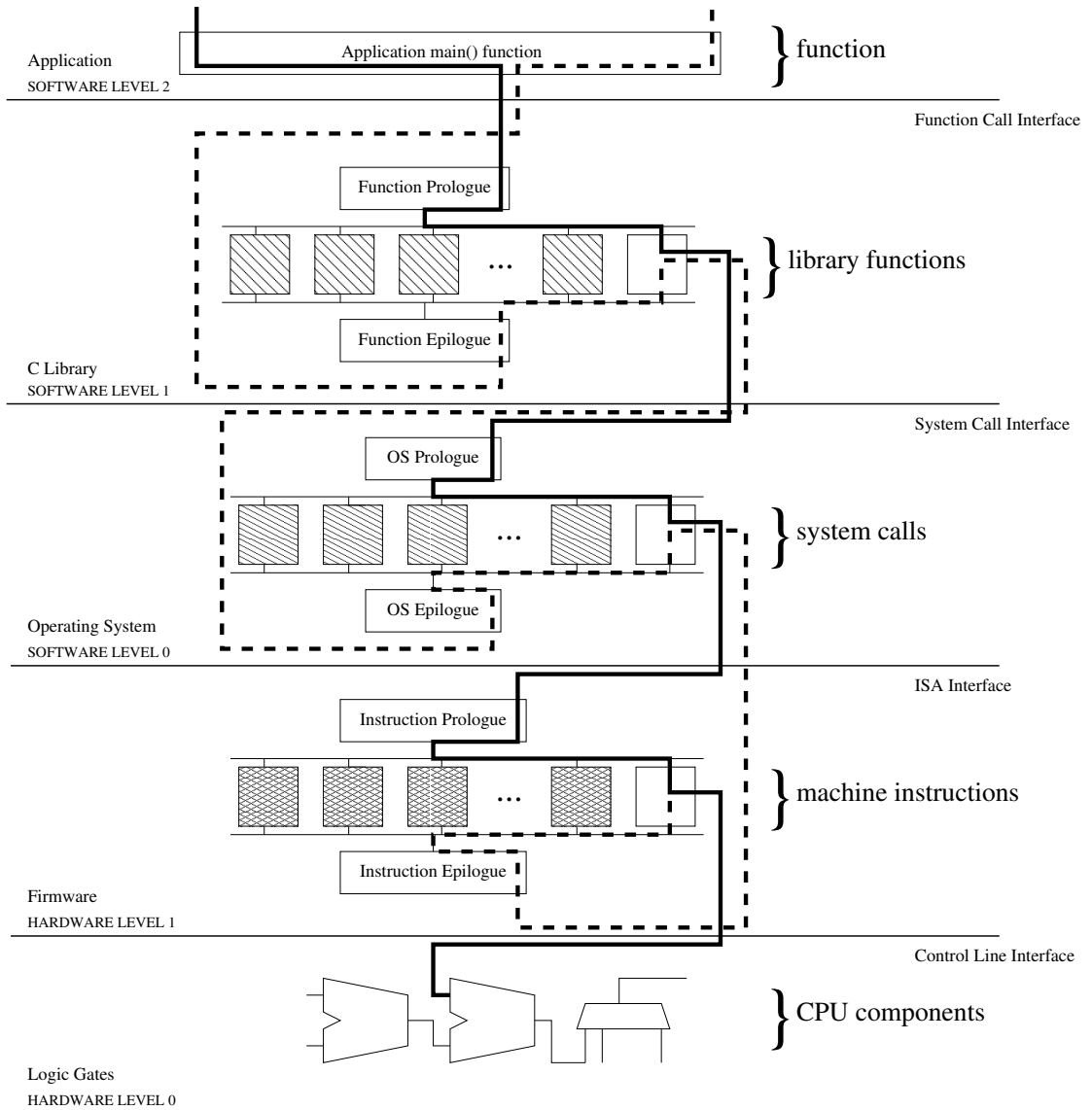


Figure 2.29: An architecture represented as a multi-level structure, based on a diagram in [242]. The solid black line illustrates a call path that passes through every level of the system: the dashed black line indicates the return journey.

Levels	Example of Operation	Prologue/Epilogue Penalty Is Due To...
SW 2 to 1	send function call	function call stack operations e.g. register values saved in memory
SW 1 to 0	sys_write system call	state saving, table lookup, function call
SW 0 to HW 1	CPU instruction	decode time, fetch time, pipeline effects
HW 1 to HW 0	Microinstruction	hardware limitations

Table 2.8: Examples of penalties incurred by crossing level boundaries.

The only way to eliminate the effects of the penalty is to move operations to a lower level. This is called *vertical migration*. Although vertical migration is not a commonly used term, the principle is widely used.

For example, vertical migration is used by many compilers as they carry out function inlining. Suppose that function A calls function B a total of n times within a loop. Generally, this will incur $n+1$ function call penalties: n for the calls to B , and one more for the call to A . By copying the code in B directly into A , the overhead is reduced to exactly one function call penalty. (This optimisation is very effective and frequently used by compilers.) Another example is JIT compilation, where the overhead of interpreting a bytecode language such as Java is eliminated by translating the bytecodes into machine code for the host ISA.

Similarly, executing every machine code instruction introduces an overhead. The use of a writable control store enables this overhead to be reduced, because the effects of n machine code instructions can be carried out by a single microprogram. The only overhead is the cost of starting that microprogram in the first place - in other words, the overhead of running a single instruction rather than n instructions.

Vertical migration can be applied to reduce either ACET or WCET. It does not necessarily involve any dynamic operations, so the time saved by migration can be calculated. For example, when vertical migration is applied within an ASIP, multiple machine code instructions are carried out by a custom piece of hardware within the CPU. The time taken by this hardware is predictable. The same is true of RFUs as described in section 2.6.7.

But RFUs also offer a way to overcome some of the issues introduced by the use of microprograms. Suppose that the microprogram within a CPU is able to control an RFU in addition to other CPU components. Adding this special type of functional unit brings the following advantages:

- RFUs can be designed as ideal targets for compilation, in which operations have no side effects.

It is common practice to build CPUs to match the capabilities of compilers: RISC CPUs are classic examples of this design strategy. The same idea can be applied in RFU design. PRISC [213], Chimaera [293], and ReRisc [264] are all designed to run automatically generated configurations, although compilers have not been developed for all three. Each operation is orthogonal to the others, which simplifies automatic generation of configurations. Applying this design within a microprogrammed CPU would simplify generation of microprograms, just as a RISC CPU design simplifies machine code generation.

This principle could have been applied within any classical microprogrammed CPU, but commercial CPU design is generally focused on optimised execution of machine code rather than supporting automatic microprogram generation. Today, the low cost of hardware implementation platforms such as FPGAs enable entirely custom CPU designs to be built without the need for a costly fabrication process (section 2.5.2), so CPU/RFU combinations can be constructed with the specific aim of permitting user microprogramming.

- RFU designs can also be focused on compatibility with future designs. Chimaera [293] is an example in which the RFU array can be extended without losing compatibility with designs that use a smaller array. This could enable the same microprogram to be used on a variety of CPUs with different instances of the same RFU.

Another way to avoid the compatibility problem is to generate both the CPU design and

the microprogram generator at the same time. This is enabled by reconfigurable hardware, specifically FPGAs, and is the method used to ensure compatibility between an ASIP and its compiler. Code compiled for one ASIP will not work on another, because custom instructions vary between instances of the design. But this does not create a significant compatibility problem because the CPU design and the compiler are generated at the same time from a single specification.

In effect, an RFU can replace the functional units within a CPU with a new component that is designed to be automatically programmed. While this does provide solutions to two of the problems associated with user microprogramming, it still does not solve the global compaction problem. Chimaera and PRISC can only implement the functionality of individual basic blocks, because instructions implemented using these RFUs must complete in a single clock cycle. (This also limits the complexity of the basic blocks that can be supported, although long basic blocks can be split into multiple instructions.) ReRisc is more advanced, but the compiler still appears to be limited to operations within a basic block [264]. This suggests that current RFUs are limited to local compaction only.

However, a solution to the global compaction problem was proposed by Fisher [85]. This solution allows ILP to be exploited across basic block boundaries without needing to make use of any backtracking search. Fisher's *trace scheduler* compacts operations along the most likely path, and adds extra code to compensate for splits and joins. This reduces program ACETs because the most common paths are effectively optimised by the process. Trace scheduling was originally applied to microprogram compaction, but subsequent applications have included code generation for VLIW and EPIC CPUs [87].

Since solutions to the various issues posed by user microprogramming exist within the literature, and since vertical migration can be applied to reduce WCET, it seems likely that the application of recent results to user microprogramming problems will be helpful. It appears that an RFU and a user microprogrammable CPU are closely related, providing the same vertical migration capability. There is no reason why an RFU could not also be programmed using a global scheduler. This would provide both ACET and WCET reductions thanks to the predictable operation of an RFU.

2.7 Summary

The literature review has examined the fields of WCET analysis, the WCET analysis difficulties caused by complex CPU architectures, and previous work that has attempted to resolve these issues. It is clear that there is a general tradeoff between performance and ease of analysis, and complex CPU architectures generally favour minimising ACET over minimising WCET or simplifying analysis. This motivates new approaches which explicitly reduce WCET without significantly increasing analysis difficulties.

The literature review has also examined the properties of execution time reduction approaches for pure software and hybrid hardware/software systems. The need for scalability is also known to be important (section 1.1), and not all of these approaches can support both scalability and ease of timing analysis. This narrows down the field of suitable options.

However, RFUs appear to meet the requirements for amenability to timing analysis, reduction of WCETs and scalability. Techniques developed for microprogram generation can be used to program RFUs from any programming language including machine code. Timing analysis is facilitated since

2.7. Summary

RFU operation can be highly predictable. And scalability is possible since a RFU configuration store can be reloaded with task- or function-specific code.

Chapter 3

Motivation and Hypothesis

Current CPU technology is not adequate for the needs of embedded hard real-time systems, because all high performance CPU implementations are subject to one or more of the following problems:

- **Timing analysis difficulties**, making determination of WCET values either
 - **pessimistic**, tending towards over-estimation due to the need to account for the behaviour of dynamic optimisation heuristics and (in some cases) timing anomalies [275]; or
 - **costly**, either in terms of CPU time or developer time, because exhaustive measurement is required [35] or a specialised and accurate CPU model is needed [121].

Examples: almost all modern CPUs are unsuitable for hard real-time systems implementation because of these problems. Modelling difficulties for real CPUs are described by Heckmann [121]. Numerous attempts to reduce pessimism exist in the literature, e.g. [159, 119, 183], but pessimism can only be reduced safely with accurate modelling of all CPU behaviour, which is costly. Even the VISA approach [9] involves some pessimism due to a continued requirement for cache analysis.

- **Poor performance** of CPU-based approaches that are amenable to timing analysis.

Examples: simple CPUs and single-path execution approaches [61] do not introduce the timing analysis difficulties described above, but performance is reduced. Using an instruction scratchpad [273, 244, 202] or a locked cache [80] does allow WCET to be reduced predictably by minimising the effects of the memory bottleneck, but provides no solution to the problem of exploiting ILP without adding complexity to analysis.

- **Poor scalability** of other approaches that can achieve high performance and are amenable to timing analysis.

Examples: co-processors can have excellent performance, and execution time can be determined easily, but the number of co-processors that can be included in a system is limited. The example of classical co-design (section 2.6.4) suggests that it is not generally possible to migrate software source code into co-processor implementations in order to reduce WCET.

The recent trend in hard real-time systems research has been towards CPUs that are designed to simplify analysis [20, 9, 69, 204, 61]. The motivation for this is clear: analysis methods are inadequate for complex CPUs. Complex CPUs are so focused on improving ACET that WCET is

not considered. The result is that analysis methods are forced to model many special cases and complex features in order to gain useful results, and the methods cannot be easily applied to new CPUs because CPU complexity is always increasing. As Edwards and Lee suggest in [69]:

“it is time for a new era of processors whose temporal behaviour is as easily controlled as their logical function.”

3.1 Revised Thesis Aims

As stated in section 1.1, the aim of this work is to characterise an architecture for a new CPU, exploring a new way to combine performance improvements and scalability without loss of crucial properties that guarantee ease of timing analysis. Here are some observations:

- The literature indicates that IPET is the best known WCET analysis approach, as it has been proved [207] to produce exact WCET values given all constraints on program behaviour, and it can incorporate behavioural constraints of any complexity. Its advantages are so great that IPET-like constraint solving approaches have been adopted by other WCET analysis techniques such as AI [280].
- The disadvantage of IPET is that it is not suitable for modelling CPU components with complex behaviour, such as pipelines and caches, because the extra constraints needed to model such components make the integer linear programming problem intractable. Although low-level CPU behaviour has been modelled using IPET (e.g. [160]), the models do not appear to be able to scale to large programs or more complex components [183, 280]. Therefore, although IPET models are able to represent some dynamic CPU components, such representation is considered to be undesirable because it reduces the scalability of the technique.

Given these, it is now possible to state the “amenability to timing analysis” requirement of section 1.1 in exact terms:

Basic block execution times, expressed in clock cycles, must be independent of execution history in order to enable the best WCET analysis techniques to be used, specifically IPET, and eliminate serious issues such as timing anomalies.

This requirement has two side effects:

- Inter-task interference effects are eliminated, because basic blocks take the same number of clock cycles even if task switching has taken place. The interference problem has been previously addressed by cache partitioning [182].
- The requirement also *appears* to prevent (1) ILP being exploited between basic blocks, and (2) the use of caches that can be changed at runtime. In other words, the instruction rate and memory bottleneck appear to be reintroduced. To avoid loss of performance, alternative solutions must be found for these problems.

The literature review has already identified some possibilities. RTR could be used to address the instruction rate bottleneck by migrating worst case (WC) paths into hardware, where timing is predictable and parallelism can be exploited. As proposed in previous work, the memory bottleneck could be addressed by locked caches (described in [15, 80]) or scratch-pads (described in [244, 202, 273]),

The aims from section 1.1 for the new CPU architecture are revised as follows:

- **Amenability to timing analysis by IPET.**

This aim is satisfied by the basic block timing invariance requirement. When basic block timing invariance is assumed and a restricted set of constraints is used, the integer linear programming problem is equivalent to a network flow problem according to [159]. Thus, in this special case, IPET is scalable.

- **Reduction of worst-case execution time.**

WCETs of programs running on the new CPU architecture will be lower than WCETs of the same programs running on comparable architectures from previous work. This will be achieved through the use of one or more of the following: application-specific RTR co-processors, locked caches, software optimisation, ASIP-style custom instructions, scratchpads, and other techniques identified in the literature.

- **Structural scalability.**

The CPU architecture will be extensible to support more tasks and higher system complexity. This aim is satisfied by allowing new configurations to be loaded at runtime.

An additional implicit requirement is that the CPU architecture should be suitable for running general software programs, written in any software language. This requirement is stated explicitly because some approaches [20, 260, 19, 1] avoid the difficulties posed by WCET analysis of CPUs by migrating software tasks into hardware platforms, but this forces different languages to be used [106]. This is not suitable for all embedded real-time systems, because it forces a partial rewrite of a program in a different language, and the characteristics of that language might not be well suited to the task at hand.

In software, the aim of reducing WCET can be satisfied in one of two ways: either reduce the execution time of all code (a *global* approach), or reduce the execution time of certain parts of the WC path (a *local* approach).

Global reductions in execution time have been introduced by some CPU features, such as simple pipelining and Harvard-style architectures, but most practical performance improvement mechanisms are only pseudo-global. For example, introducing a cache does not reduce the execution time of every possible path through a program. It usually reduces the program ACET because the execution time of the most likely paths is reduced, but less likely paths may not benefit. The same is true for superscalar out-of-order execution. Such pseudo-global execution time reductions are not useful for WCET reduction because they involve dynamic optimisations which are known to be difficult to model. But global execution time reductions are also not likely to be useful because they are limited by the memory bottleneck and the instruction rate bottleneck.

The remaining possibility is the use of local optimisations to reduce the length of the WC path (section 2.3.9) within programs. This has the advantage of being focused on the parts of the program that contribute most to the WCET, which are identified as a side-effect of IPET and other WCET analysis techniques. Local WCET reductions have previously been applied using scratchpads [244] and locked instruction caches [80], and a key feature of such approaches is that the WC path must be re-identified after each taken decision, because each decision may change the WC path. Failing to do this results in a suboptimal “single-path analysis” as described by Falk [80].

The WCET reduction process is a cycle of identifying and optimising WC paths until no more local optimisations are possible (Figures 2.9 and 2.10).

The implication of this is that the WC path optimisation process must be very efficient, because it forms part of the objective function of a search process. Furthermore, it is not possible to evaluate the effects of an optimisation in isolation and then meaningfully combine multiple optimisations into a good solution, so it may be necessary to try very similar optimisations repeatedly in slightly different conditions. This forces a new requirement:

The process of evaluating a possible optimisation for a worst-case path must make use of algorithms that are as efficient as possible so that WC path optimisations can form part of a larger search process.

3.2 Statement Of Requirements

At this point, the aims of this work have been reexamined based on the literature, and two new requirements have been identified. The complete set of requirements for the new CPU architecture can now be restated as follows:

1. **Full support for general software programs**, with compiler and language independence, so that achieving a WCET reduction does not require even a partial rewrite in a new language.
2. **Basic block timing invariance**, to ensure amenability to timing analysis by IPET.
3. **WCET reduction** of programs to a greater extent than earlier predictable architectures.
4. **Efficient optimisation process**, to support WCET reduction algorithms that operate by a search process.
5. **Scalability** to permit WCET reductions in programs of any size.

3.3 Approach

The high-level requirements for the new CPU architecture are known, but the strategy to be used to achieve them is not. The key requirement is “WCET reduction versus previous work”. If it were not for this requirement, then a known solution such as the use of locked caches [80] or an instruction scratchpad that is updated during execution [240] could be applied to the problem. The WCET reductions must go further than whatever is possible using these approaches.

Fundamentally, this is a matter of exploiting ILP in software: the one way of increasing the speed of program execution that is (a) applicable to general programs and (b) not currently supported effectively by processors that facilitate timing analysis. Although locked caches and scratchpads provide a solution to the memory bottleneck, they don’t provide a solution to the instruction rate bottleneck, because exploiting ILP using current solutions will make timing analysis more difficult. (This is why VISA [9] does not make use of ILP in its simple mode.) In [217], Rochange proposes a CPU that exploits ILP within a single basic block: this form of ILP preserves basic block timing invariance, but the possible WCET reduction is limited by branches [186, 268].

Therefore, to go further than previous work, ILP must be exploited across basic block boundaries. This could achieve the same execution time reductions observed within superscalar and

VLIW CPUs, but without dynamic optimisation or caching. Exploiting ILP across basic block boundaries would seem to be incompatible with the requirement for basic block timing invariance: for example, how can speculative execution be carried out?

RTR provides a possible answer. Custom logic could be used to reimplement parts of a WC path within a program while leaving the original basic blocks intact for use by other paths. The results of this process would still be composed of basic blocks. Provided that the transformations could be modelled within IPET, this would allow the ILP within code to be exploited while preserving ease of analysis.

3.4 Hypothesis

Run-time reconfigurable (RTR) hardware can be used to exploit instruction level parallelism (ILP) within software in order to reduce the worst case execution time (WCET) of a program without adding significant complexity to the WCET analysis process. The application of this technique will reduce the worst-case effects of the instruction rate bottleneck (identified earlier) to a greater extent than previous work in the area of predictable CPU architectures, these being CPU designs that are specifically intended to support WCET analysis. The technique will also be demonstrated to scale to support large programs.

3.5 Evaluation Criteria

The work will be evaluated against the following criteria:

- Is the hypothesis (section 3.4) demonstrated?
- Is an implementation of the CPU architecture functionally correct?
- Does an implementation satisfy the requirements listed in section 3.2?

The evaluation will be performed by a mixture of analysis, written proof, and implementation followed by experimentation. An experimental model is effectively a requirement due to the high difficulty of reasoning about program and architectural properties without a working model.

Chapter 4

Architecture

Chapter 3 stated the requirements for the new CPU architecture:

1. Full support for general software programs, with compiler and language independence, so that achieving a WCET reduction does not require even a partial rewrite in a new language.
2. Basic block timing invariance, to ensure amenability to timing analysis by IPET.
3. WCET reduction of programs to a greater extent than earlier predictable architectures.
4. Efficient optimisation process, to support WCET reduction algorithms that operate by a search process.
5. Scalability to permit WCET reductions in programs of any size.

Together, these requirements suggest a simple CPU combined with an extension that provides a mechanism for WCET reduction. As section 3.3 suggests, this extension could take the form of a *run-time reconfigurable* (RTR) module of some sort, because this would allow the exploitation of *instruction-level parallelism* (ILP) in code without losing the benefit of simple analysis. In conjunction with a WCET reduction process for identifying WC paths, this would demonstrate the hypothesis (section 3.4).

4.1 Introducing TARGET and the WCET Reduction Process

In this chapter, the CPU architecture is given the name “TARGET”. Previous work is used to create a specification for TARGET in section 4.2, and this specification is then evaluated against the requirements in section 4.3. This chapter avoids details such as the *instruction set architecture* (ISA) and the set of ALU functions: these are regarded as architectural parameters, to be finalised by an implementation.

This chapter meets one of the goals of this work, by characterising a new CPU architecture that meets the requirements listed above. However, two other components are required before TARGET can be evaluated: (1) a working implementation of TARGET, and (2) a WCET reduction process. The first component is discussed in chapters 5 and 6, which describe the process of implementing TARGET. The second component is shown in Figure 4.1: this design is refined as details of TARGET and its implementations are introduced, and finally implemented in chapter 7.

Figure 4.1 divides the WCET reduction process into four conceptual levels. At the source level, a complete program is produced in machine code for the TARGET architecture. The requirements

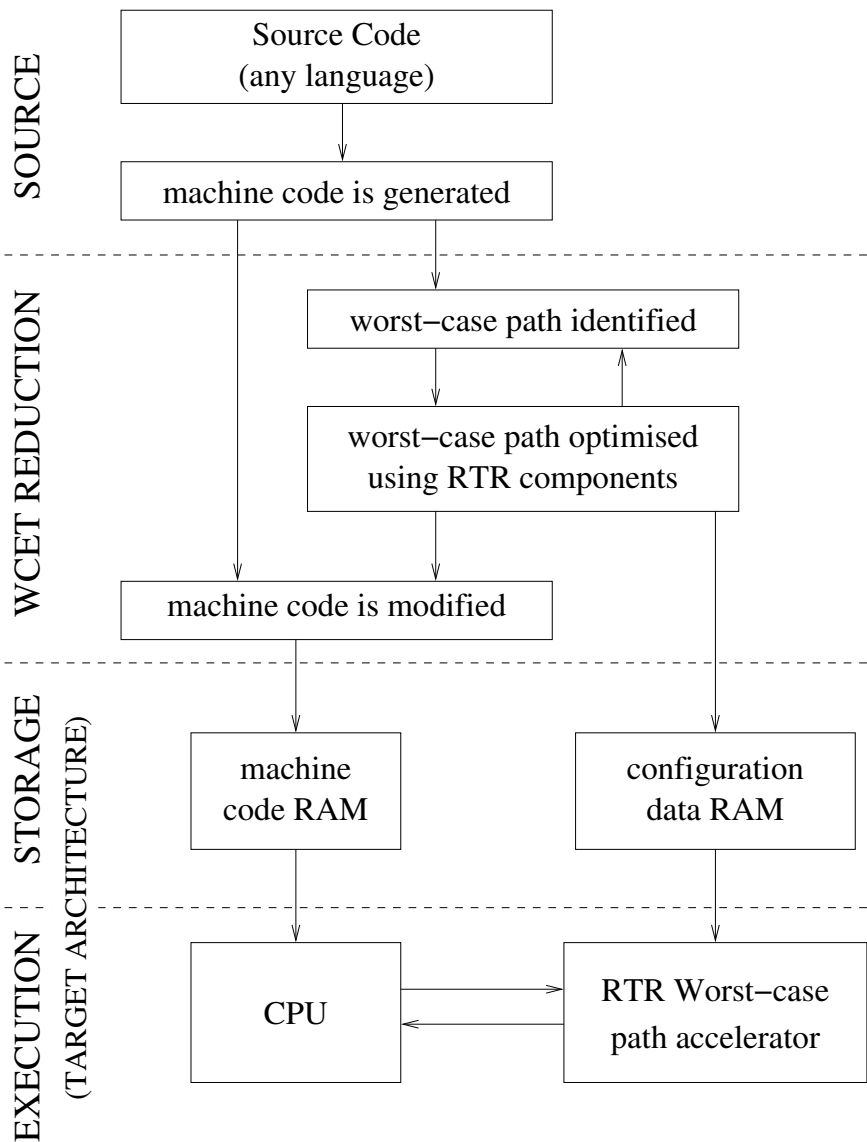


Figure 4.1: The four conceptual levels of the WCET reduction process, in abstract form.

say that general software programs must be supported, so there can be no inherent dependence on any particular software language. At the WCET reduction level, an optimisation process is applied to reduce the execution time of the WC path using RTR components. At the storage level, there is an arrangement of memory elements for storing program instructions in two formats: RTR configurations and machine code. At the execution level, the CPU executes machine code and optimised worst-case paths. This chapter is concerned with the storage and execution levels only.

4.2 Construction of TARGET Architecture

This section describes the process used to define the architectural parameters of TARGET. The literature in section 2.6 indicates that a user microprogrammed RFU would meet the requirements for WCET reduction. This is the “RTR Worst-case path accelerator” in Figure 4.1: an array of functional units controlled by microcode. The following sections describe the process of transform-

ing a simple CPU architecture into TARGET, which is a simple CPU with RFU extensions that meets the requirements (section 3.2). It is capable of supporting the WCET reduction process and demonstrating the hypothesis.

4.2.1 Starting Point - CPU

The best starting point for the work is a simple CPU that satisfies all of the requirements except for WCET reduction. An RFU controlled by a microprogram store is added to this design to enable WCET reduction. The requirements for this simple CPU make it suitable for (1) IPET analysis [159], and (2) any program:

1. Basic block execution time, in clock cycles, is independent of execution history;
2. No limit on the number of tasks or the complexity of each task.

Because of engineering costs, it is not a good idea to design a CPU entirely from scratch unless there is a specific reason to do so. However, two reasons exist: (1) the design can be focused on the efficiency of both the RFU and the CPU, which is potentially better than focusing first on optimising CPU design, then adding the RFU; and (2) the RFU can be designed to support efficient microcode generation.

But defining a new CPU introduces other problems. An entirely new ISA forces a new code generator to be written, along with a new assembler and linker. It is possible to modify a free tool chain such as `gcc`, but this effort is time consuming. Fortunately, all ISAs are abstract definitions. ISAs do not normally specify exact timing for execution (VISA [9] is an exception), so virtually all ISAs meet requirement 1 in principle. Additionally, any ISA that supports a sufficient address space meets requirement 2: this includes all 32-bit ISAs.

Therefore, while the TARGET architecture should start from scratch, it should also adopt an existing ISA. The simplest ISAs are RISC-like ISAs [144, 195], with a minimal set of instructions and a single addressing mode. Consequently, these ISAs are easy to implement on minimal hardware. Many soft CPU cores used within FPGAs implement a RISC ISA such as Microblaze [286] or OpenRISC [150]: these are similar in principle to classic RISC ISAs such as MIPS.

However, an adoption of a RISC-like ISA should not force the use of a RISC-like microarchitecture. RISC CPUs interleave the execution of multiple operations within a pipeline, involving five stages in classical designs [195], which is undesirable in this case because it may affect the basic block timing invariance property. An alternative is the cyclic execution paradigm used in classical CISC architectures such as 8086 [230] and Motorola 68000 [181], which repeatedly fetch an instruction then execute it. The benefits of this scheme, which is not used in modern complex CPUs, are:

- Simple implementation with minimal resources required;
- One register memory can be used for general-purpose registers and for special-purpose registers such as the *program counter* (PC);
- One ALU is used both for instruction execution and incrementing the program counter, so less logic is used;
- No pipeline hazards to be considered;

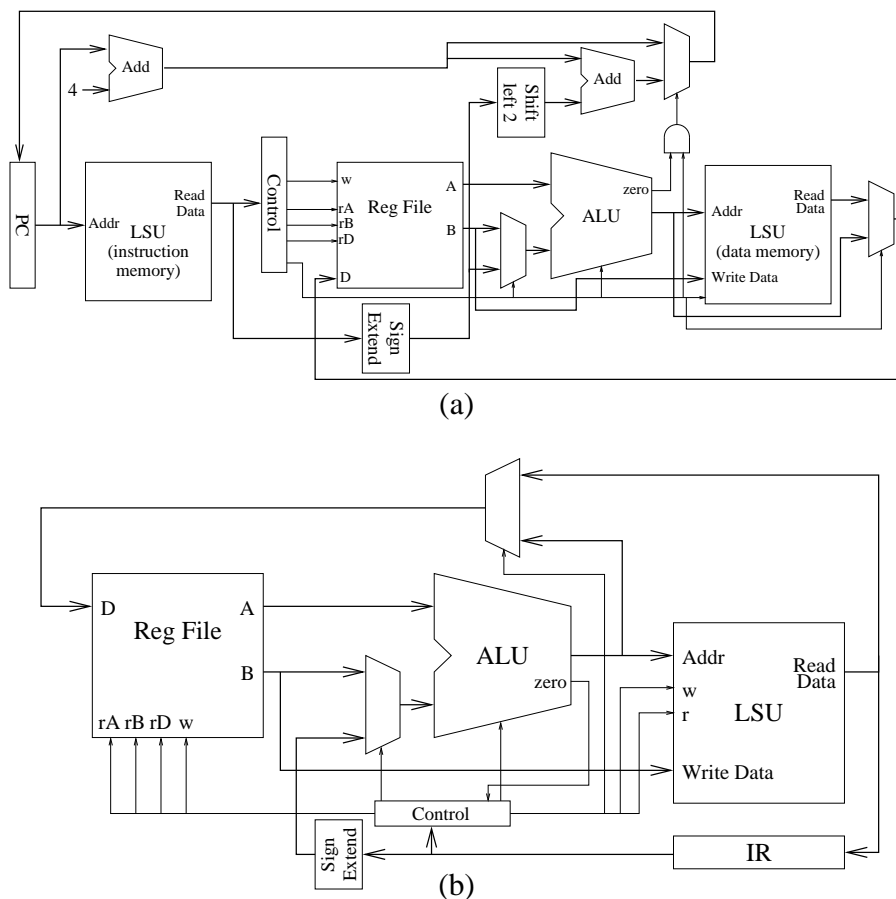


Figure 4.2: (a) A simple CPU data path from [195], Figure 5.19. (b) Further simplifications for a cyclic execution CPU.

- Each instruction has a fixed execution time;
- Simple control using either a state machine or a microprogram.

When the benefit of a cache is lost, the CPU speed is bounded by memory speed. In this environment, there is no need for a pipeline.

In [195], Patterson and Hennessy provide an example of a simple CPU data path for executing RISC-like instructions (Figure 4.2(a)). This arrangement has many of the properties required, but it has been designed with the assumption that one instruction will be executed every clock cycle. Therefore, some units have been replicated. The program counter has its own dedicated data path involving two adders. There are also two memory units for access to instructions and data. For cyclic execution, further simplification is possible, as the same data path hardware can be used for both the program counter and general-purpose registers, and the same memory bus can be used for both instructions and data. This means that there is no need to replicate any of the units: the operation of each unit can be time-multiplexed.

Figure 4.2(b) shows the equivalent of Patterson and Hennessy's simple data path for a cyclic execution CPU, with the ALU, register file, and *load/store unit* (LSU) shown as used in [195]. Note that the PC register has been subsumed into the main register file. The simple data path is able to carry out the following operations:

1. Compute $rD \leftarrow rA \text{ aop } rB$ for any registers rD , rA and rB , and any ALU operation aop ;
2. Compute $rD \leftarrow rA \text{ aop } i$ for any registers rD and rA , and a constant i from the *instruction register* (IR);
3. Load memory from address $rA + i$ into rD ;
4. Store memory from register rD at address $rA + i$;
5. Load memory from address rA into IR.

These low-level operations are sufficient to implement a RISC-like ISA on a cyclic execution data path like Figure 4.2(b), given an appropriate control unit. They support instruction fetch (using op 5), PC increment (using op 1 with a constant rB), and the following classes of RISC instruction:

- ALU instructions such as `add` and `addi` (using operations 1 and 2);
- Branch and jump instructions (also using operations 1 and 2);
- Memory access instructions (using operations 3 and 4).

Addition of a suitable control unit enables conditional execution for branches, which make use of the ALU for both PC modification and data comparisons. The only remaining class of RISC instructions is the “system” class of operations that modify CPU flags (e.g. `l.mtspr` on OpenRISC [150]), which is handled as a special case.

4.2.2 A Simple CPU, plus an RFU

Having specified a class of CPU design, the definition of TARGET continues with RFU extensions to that design. The RFU requirements have been previously identified as follows:

1. Preserve basic block timing invariance (section 3.2),
2. Support for implementing hotspots in general, and fragments of the WC path in particular (section 2.6.7),
3. Support for efficient microcode generation (section 2.6.8).

Consider Figure 4.3. Three components of the simple CPU architecture (ALU, register file, and multiplexer) are shown on the left hand side. The right hand side shows an abstract view of the same components in array configurations: a 1×1 array, a $1 \times w$ array, and an $h \times w$ array.

The 1×1 array is the arrangement in an abstract simple CPU. The effectiveness of this arrangement has been proved by countless CPU designs, but it is limited to sequential code and thus cannot exploit ILP.

The $1 \times w$ array is the arrangement in an abstract superscalar, VLIW or EPIC CPU (section 2.2.2), which contains w ALUs able to operate in parallel. In this arrangement, the multiplexer acts as an interconnect, and the register file may be distributed across all array elements or local to clusters of functional units [87]. This arrangement has proved an effective way to exploit ILP in code by many CPU designs, although there is no point in increasing w beyond the limit of ILP within the software being executed [268], because the additional functional units will not be used. Because w may be quite small (4 to 8 units appears to be common), it is possible to allow each multiplexer to

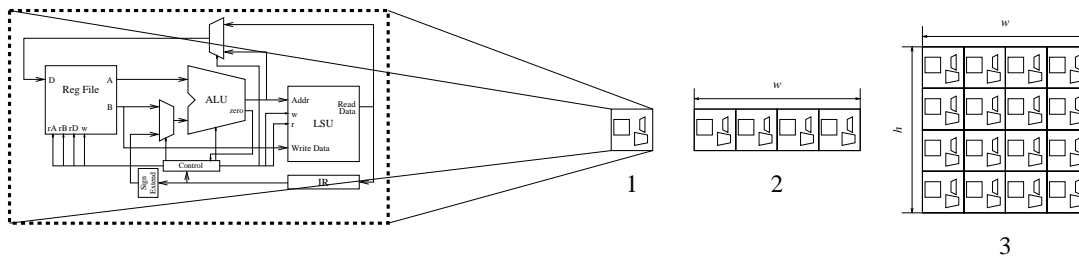


Figure 4.3: The data path from Figure 4.2(b), shown on the left, is repeated 1×1 times in a simple CPU, $1 \times w$ times in a superscalar CPU, and $h \times w$ times in a CGRA.

select input from any other unit, so that the array is fully connected. Additionally, efficient algorithms exist to allow ILP to be exploited on such arrays, both for dynamic operation [236, 254], and for VLIW code generation [85, 87].

The $h \times w$ array is the arrangement in an abstract CGRA, which contains hw ALUs able to operate in parallel. The effectiveness of this arrangement has been shown by various CGRA designs (section 2.6.6). But there are two problems. Firstly, general software programs cannot be efficiently mapped onto such a large number of units, because software languages are not an adequate way to make use of the parallelism that is available. (This is why ALE-X [113] and DIL [101] have been proposed as ways to specify CGRA configurations.) Secondly, because the number of units is so large, the interconnect must be restricted so that arbitrary connections are not possible. This forces the use of heuristic place and route searches. The latter problem can be solved by linear interconnection layouts to support the use of non-backtracking allocation algorithms, as demonstrated by Piperench [40], but these do not address the first problem.

Since the purpose of the RFU in TARGET is to reduce the WCET of software by exploiting ILP, a full $h \times w$ CGRA is not suitable. Even if a Piperench-like arrangement was used to allow configurations to be evaluated quickly, the co-design problem described in [106] would still be an issue, because parts of the WC path would need to be expressed both in C (for maximum efficiency of software implementation) and in a specialist CGRA programming language. This would rule out automatic search for a WCET reduction by forcing the programmer to explicitly rewrite some parts of the program in a second type of language.

Therefore, a $1 \times w$ array should be used as the RFU. This meets all three RFU requirements:

1. Preserve basic block timing invariance. This property is maintained by avoiding any dynamic operation in the array. By using microprogramming (section 2.6.8) to control the RFU, complex application-specific instructions can be implemented by the RFU using only static operations.
2. Support for implementing hotspots in general, and parts of a WC path in particular. This property is maintained by choosing an array size to support the implementation of software. The $1 \times w$ array shape supports exploitation of ILP in general code, and although the amount of ILP available will vary [268], there is no requirement for a specialist language.
3. Support for efficient microcode generation. This requirement is also supported by the $1 \times w$ array shape, as configurations can be generated efficiently using known algorithms such

as [85].

Suppose that all members within the $1 \times w$ array include an ALU and register file, and thus may implement arithmetic functions and store information. Two additional features are optional: an LSU for memory access, and hardware required to enable CPU functionality (e.g. instruction register and dispatch logic). This allows RFU elements to be divided into three classes:

- **Basic element** - contains only RFU functionality, which can be defined as the functions available using a register file and an ALU. Let there be l such elements in the TARGET architecture.
- **Memory access element** - contains RFU functionality (register file and ALU) plus an LSU connected to some external memory. Let there be m such elements in the TARGET architecture. These elements can carry out arithmetic operations and access memory. Memory must respond deterministically, without an unpredictable variation in latency. Therefore, caches cannot be used, and burst memory transactions (section 2.2.1) should be avoided. Scratchpads [273, 244, 202] are likely to be a good way to implement memory in TARGET.
- **CPU element** - contains RFU functionality, an LSU, and hardware to support CPU functionality. Let there be n such elements in the TARGET architecture. These elements can carry out arithmetic operations, access memory, and execute machine instructions.

Figure 4.4 illustrates an architecture composed of l , m and n copies of each of these elements. This is a high-level view of TARGET. In total, there are $l + m + n$ elements. The array shape is $1 \times (l + m + n)$.

A major benefit of the modular design is that all RFU elements share the same type of control store. This is important for microprogramming support, which will be used to control the RFU in place of the dynamic control scheme that would be used in a superscalar CPU. A modular design helps to support user microprogramming by ensuring that the architecture uses microcode in a consistent way, free of side effects and non-orthogonal functions which complicate the compilation process.

4.2.3 Communications Infrastructure

The interconnect scheme (Figure 4.4) is not defined by the architecture. By using different arrangements, different types of CGRA can be represented. For a small value of $l + m + n$, a totally connected arrangement can be used. This will not scale to large numbers of RFU elements, because wiring is a limiting factor in CPU design. According to [87],

“[Today,] designs are limited by wiring and interconnection congestion. In general, structures requiring global interconnection or many wires, such as register files and bypass/forwarding networks, are expensive in today’s silicon processes.”

In the TARGET architecture, the register file size can be easily restricted to an efficient size. The area of a register file is $O(p^2)$ for p ports [87], so small register files can be made by restricting the number of ports. This is appropriate for the architecture shown in Figure 4.4 which specifies a series of separate register files, one per RFU element.

However, the interconnect linking register files to ALUs (Figure 4.4) is not so easy to restrict. If this were able to connect any ALU to any functional unit, like a crossbar switch, it would be $O(nm)$

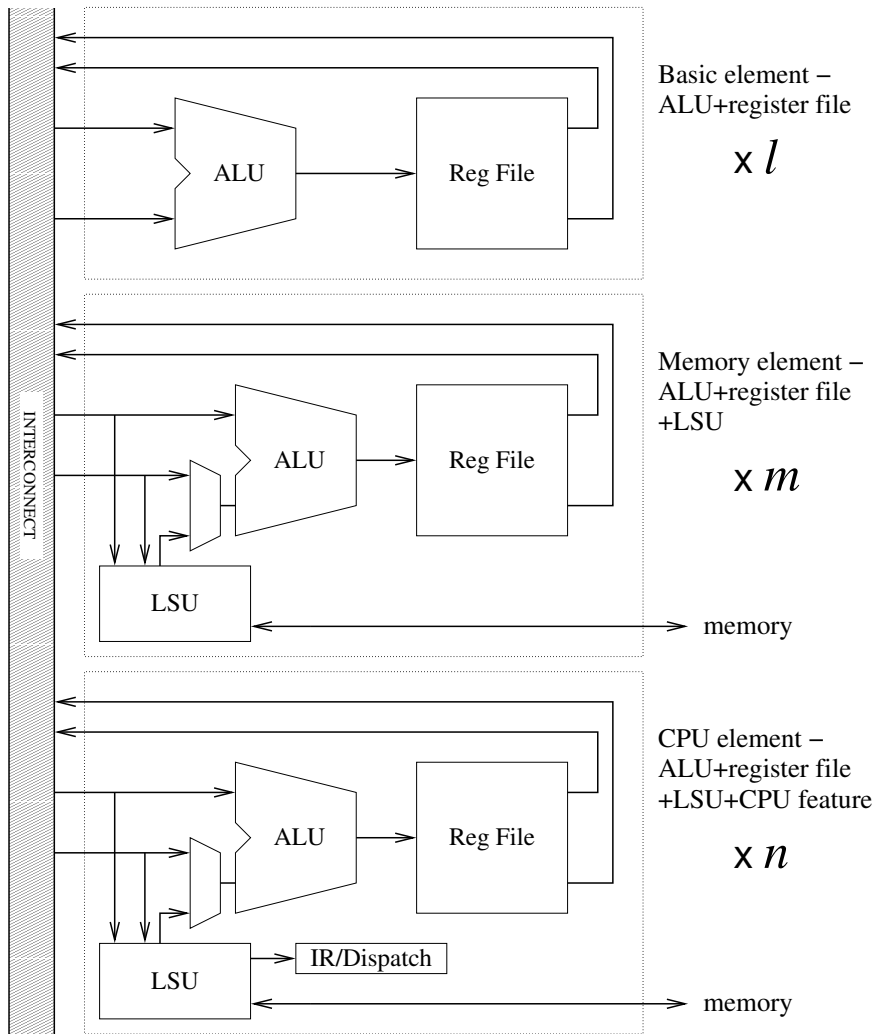


Figure 4.4: An abstract high-level view of an RFU architecture with l basic elements, m memory access elements, and n CPU elements.

in size for n inputs and m outputs. A large area is undesirable, because of the implementation cost and the effect on maximum clock frequency. But large values of n and m may be unnecessary, because the parallelism benefits of a large number of RFU elements cannot be exploited within all programs [268].

Without an implementation, it is not possible to evaluate the tradeoff between interconnect area and exploitation of ILP. It is also not easy to see what the effects of restricting a totally connected network might be on the efficiency of configuration generation algorithms. Therefore, the abstract model must permit the degree of connectedness to be restricted in arbitrary ways, including “no restriction at all”.

So TARGET makes no statement about the interconnect layout to be used. All connection arrangements are permitted with the single proviso that dynamic operations are not permitted. For instance, Ethernet-style networks on chip are not suitable because they do not operate with predictable timing. (Ethernet uses pseudo-random time delays to resolve collisions on the network bus.) However, point-to-point links and statically scheduled networks on chip such as *time division multiple access* (TDMA) buses are suitable because low-level timing properties can be guaranteed.

4.2.4 Controlling CPU Operations

The control unit defined for the TARGET architecture must include RAM to enable user microprogramming. In light of the limitations of wiring, it would appear most efficient to use a local control memory at each node, but a shared global control memory is also possible. The microprogram store memory within TARGET appears within every node in the high-level view of the architecture as shown in Figure 4.5.

The contents of the microprogram store come from two different places. Firstly, the built-in microprogram is present in each microprogram store when the CPU is activated for the first time. It uses the RFU resources to interpret machine code. Secondly, user microprograms are produced by a code generator and then uploaded by programs as required.

The built-in microprogram has to be specified as a part of the TARGET architecture. Therefore, TARGET software tools must include a program to generate the required microprogram data. But TARGET software tools must also include a program that produces an interface between user programs and the RFU. This interface is a driver for a hardware device: the TARGET RFU. It must allow software to generate RFU commands for any purpose without knowledge of the specifics of the hardware. The aim is to allow software to apply WCET reduction or other optimisations in a device-independent way.

4.2.5 TARGET Architecture Characterisation

All TARGET architectures are instances of the layout shown in Figure 4.5, i.e. with specific values for l , m and n , a specific ISA for the CPU elements, and appropriate microprograms loaded to support that ISA. These components may exist in the form of a hardware design (e.g. HDL code) or in the form of a simulator program, or both. Each instance also includes a microprogram generator matched to its microinstruction encoding. The parameters of the architecture are enumerated in Table 4.1, and TARGET is shown as a component in the abstract WCET reduction process in Figure 4.6.

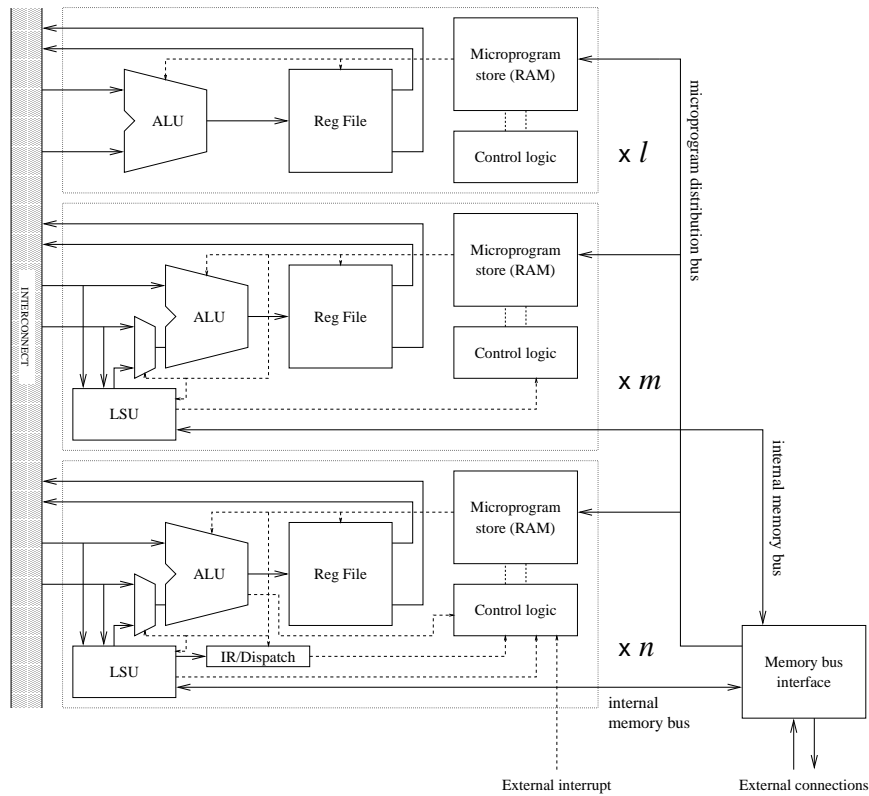


Figure 4.5: A high-level view of the TARGET architecture from Figure 4.4, with microprogram stores at each element, and a second memory bus for distribution of microprogram data.

Parameter
ALU function set
Interconnect
ISA
l, m, n values
Register file dimensions
Built-in microprogram
Microcode interface
μ code store dimensions

Table 4.1: The parameters of the architecture.

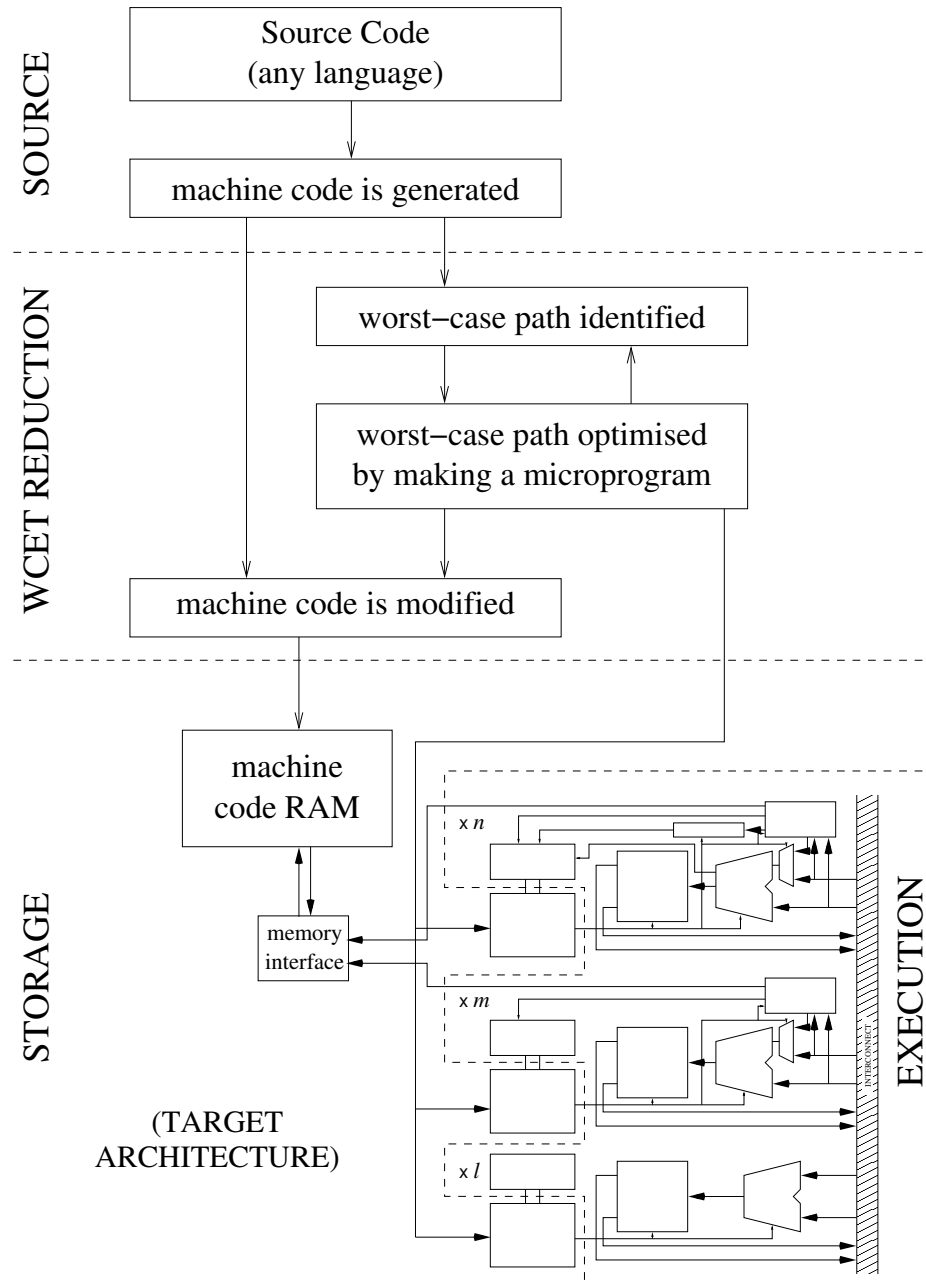


Figure 4.6: The WCET reduction process of Figure 4.1, using the TARGET architecture as shown in Figure 4.5.

4.3 Theoretical Evaluation

This section evaluates how well the TARGET architecture can meet the requirements listed in section 3.2, irrespective of any implementation of TARGET that is actually built. Each requirement is addressed in turn by sections 4.3.1 through 4.3.5.

4.3.1 Full Support For General Software Programs

No specialist language is required by TARGET. A standard machine code can be used as the input to the microcode generation process, which generates RFU configurations, and the TARGET CPU itself. Therefore, TARGET can provide WCET reduction features for any program regardless of the software language or compiler used to build it.

4.3.2 Basic Block Timing Invariance

The basic block timing invariance property can only be lost if the CPU is able to process an instruction in more than one way. For example, if an instruction cache is present, there are at least two ways to process an instruction: (1) when the instruction is in cache, and (2) when it is not.

In TARGET, there is exactly one way for each machine code instruction to be processed, because the cyclic execution process has no data-dependent state. Therefore, basic block timing invariance is assured during machine code execution. The same is true for any RFU operation because each microinstruction can also only be executed in one way.

4.3.3 Efficient Optimisation Process

This requirement specifies that the RFU code generation process should be fast. The literature includes several examples of fast algorithms targeting a $1 \times w$ array. Trace scheduling algorithms such as [85] are a good example, having been proposed and developed with the specific intention of efficiently solving the global microprogram compaction problem for $1 \times w$ arrays of functional units. But other algorithms are also suitable. Section 2.2.2 discussed some algorithms for dynamic superscalar out-of-order issue, such as [254, 236]. These could be adapted to generate RFU code statically. Being designed for run-time operation, these algorithms are also very efficient.

4.3.4 Scalability

This requirement sets TARGET apart from approaches that can only reduce the execution time of a limited number of program components. For example, ASIPs are limited by the hardware space taken up by custom instructions. One way to allow TARGET to scale to programs of any size is to treat the microprogram store as a scratchpad. As section 2.6.7 describes, this will allow the supported set of custom instructions to be changed during execution in a way that is predictable. In effect, this is run-time reconfiguration of the RFU.

This approach is scalable to any number of tasks because each task can use a different set of microprograms. Task switching is an obvious time to reload the microprogram store: the OS carries out the copy operation. Each task may also need to be divided into regions, each with their own set of microprograms. This partitioning technique has already been demonstrated using scratchpads [240, 201] and locked caches [80]. It is related to overlaying [191]. The cost of using

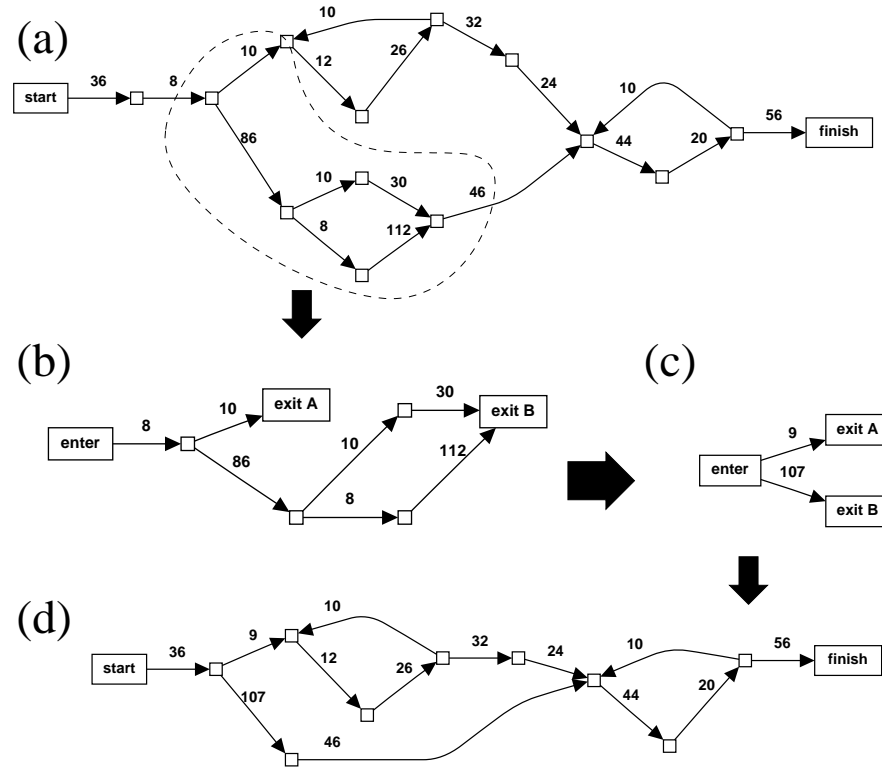


Figure 4.7: The effects of an abstract machine code \rightarrow RFU transformation. The input T-graph is (a) - edges are labelled with their execution time costs. A subgraph (b) is selected (enclosed by dashed line), and transformed, (c), to a new RFU configuration with execution time properties as shown. The effect on the T-graph is shown in (d).

this technique is that programs have to be divided into regions, but this can be an automatic process [201], and in principle it can be used with any program. Thus, TARGET is scalable, although the mechanism that permits scalability does make the assumption that partitioning is possible.

4.3.5 WCET Reduction versus Previous Work

TARGET must provide access to greater WCET reductions than previous work. Figure 4.6 indicates that this process is carried out by tools that have not yet been specified. This section describes how these tools might work, in abstract terms. Here, it is assumed that the IPET approach for WCET analysis is used to identify the WC path.

For each tool, the starting point is an IPET model of a program: specifically, a T-graph $G = (V, E)$ (section 2.1.2) plus behavioural constraints. (The IPET process is used as described in Appendix E and by [207].) IPET determines the overall WCET Z_G and the maximum number of executions $f(x)$ for each edge $x \in E$ along the WC path through the code. Edges represent basic blocks, and each edge also has an execution time cost $\gamma(x)$.

Figure 4.7(a) illustrates a T-graph for a sample program. (This is based on an example from [207].) When the IPET process is complete, the $f(x)$ values indicate the current WC path through the program. The WC path is a sequence of basic blocks representing the path through the program

that results in the highest execution time. To reduce the WCET, the WC path must be reduced first.

Since the WCET $Z_G = \sum_{x \in E} \gamma(x)f(x)$, the basic blocks that contribute most to the WCET can be identified by sorting all edges using a function of $\gamma(x)$ and $f(x)$. (The total contribution of an edge to the overall WCET is $\gamma(x)f(x)$.) This indicates the subpaths of the WC path that should be optimised first. An algorithm or heuristic can detect the best fragments of the WC path for particular types of WCET reduction optimisation. Connected groups of T-graph edges would be selected for optimisation together: typical groups might include loops and frequently-used functions, although other types would be possible. An example of such a group is shown in Figure 4.7(a) within a dashed line.

After this identification process, the group would be removed from the surrounding program and processed as a separate subgraph, such as Figure 4.7(b). A microprogram generator then converts the machine code of the edges in the group into a microprogram for the TARGET RFU. As ILP can be exploited within the group, and the overheads of machine code execution are eliminated, the execution time of each path through the group is reduced. The result is a new subgraph that is functionally identical to the first, but has reduced execution time costs. An example is illustrated by Figure 4.7(c).

The microprogram for the new subgraph replaces the original machine code, using an ASIP-like custom instruction. The IPET model is also updated to reflect the change to execution costs. An example of the effect is shown in Figure 4.7(d). The IPET process must be reapplied after each change to find new WC path fragments, because any transformation could change the maximum number of executions $f(x)$ for any edge $x \in E$.

Different microprogram generators might put different restrictions on the type of subgraph that could be optimised. For example, a type of trace scheduler known as a superblock scheduler [48] is restricted to subgraphs that are trees, i.e. with one entrance node, but any number of exits. The implication of this is that WC path fragments must be selected to match the properties of the generator that is being used.

4.3.6 TARGET Bottleneck

TARGET includes a form of memory bottleneck. Memory must respond in a predictable way to satisfy the basic block timing invariance property. Unfortunately, this makes it difficult to use memories that require burst transactions (section 2.2.1), such as the dynamic RAM modules in 2008-era PCs. Burst transactions are ideal for filling caches, but this is not useful when basic block timing invariance must be maintained. The bottleneck forces the use of scratchpads, to reduce memory latency on a predictable subset of the program, and the use of static RAM that responds predictably.

This appears to be an inherent problem in TARGET. It could be argued that the bottleneck is a natural consequence of the requirement to execute general software. Inevitably, the limits of a simple CPU will be present. One way to reduce the bottleneck is to add more memories and buses. Each of the $m + n$ units could be operating on a different memory bus simultaneously.

However, it must be noted that TARGET does not lack support for burst transactions entirely, as bursts are an ideal way to fill microprogram stores and scratchpads quickly. The requirement for exactly predictable operation only applies to the execution of code. Provided that each burst completes before a deadline, code execution will be unaffected.

4.3.7 Summary

The evidence indicates that the TARGET CPU architecture meets the requirements set out in section 3.2. It supports general software programs, has the basic block timing invariance property, can make use of efficient microcode generators, and supports scalability using a scratchpad approach (section 4.3.4). Finally, it can reduce the WCET of a program by making use of WC path optimisations. However, an implementation of TARGET will be needed for a complete evaluation.

Chapter 5

Static Implementation

Theoretical reasoning about the capabilities of the TARGET architecture is of limited use because of the number of assumptions that must be made. In the absence of a working CPU model, many properties of TARGET are simply unknown, because they are dependent on factors that are hard to predict without an implementation. Some of these are described by Table 5.1. Therefore, the goal of this chapter is to evaluate an implementation of the architecture described in chapter 4 against the requirements (section 3.2). The prototype will be implemented using an HDL and tested on an FPGA (section 2.5.3). It will provide the storage and execution levels of the WCET reduction process (Figure 5.1). The following components are needed:

- A microarchitecture in *hardware description language* (HDL) form;

Property	Reason
Basic block execution time	Depends on the basic block, the choice of ISA, the compiler, and the implementation of the ISA.
Reduction in WCET possible using an RFU as described previously	Depends on the program, the RFU configuration builder's algorithm, the choice of basic blocks for RFU implementation, and the RFU parameters.
WCET relative to other CPU cores	Depends the implementation of the ISA.
Number of RFU configurations that can be supported	Depends on the size of the microprogram RAM, the amount of microprogram space taken up by each configuration, and the amount of microprogram space taken up by the implementation of the ISA.
Size of CPU/RFU hardware	Depends on the features of the RFU and ISA that have been implemented, the size of the array, the interconnect used, and the amount of memory used for microprograms.

Table 5.1: Some of the properties of the TARGET architecture that are hard to evaluate without a working model. All of these properties are dependent on the CPU architecture parameters from Table 4.1, but the relationship between each parameter and each TARGET property is complicated by implementation details.

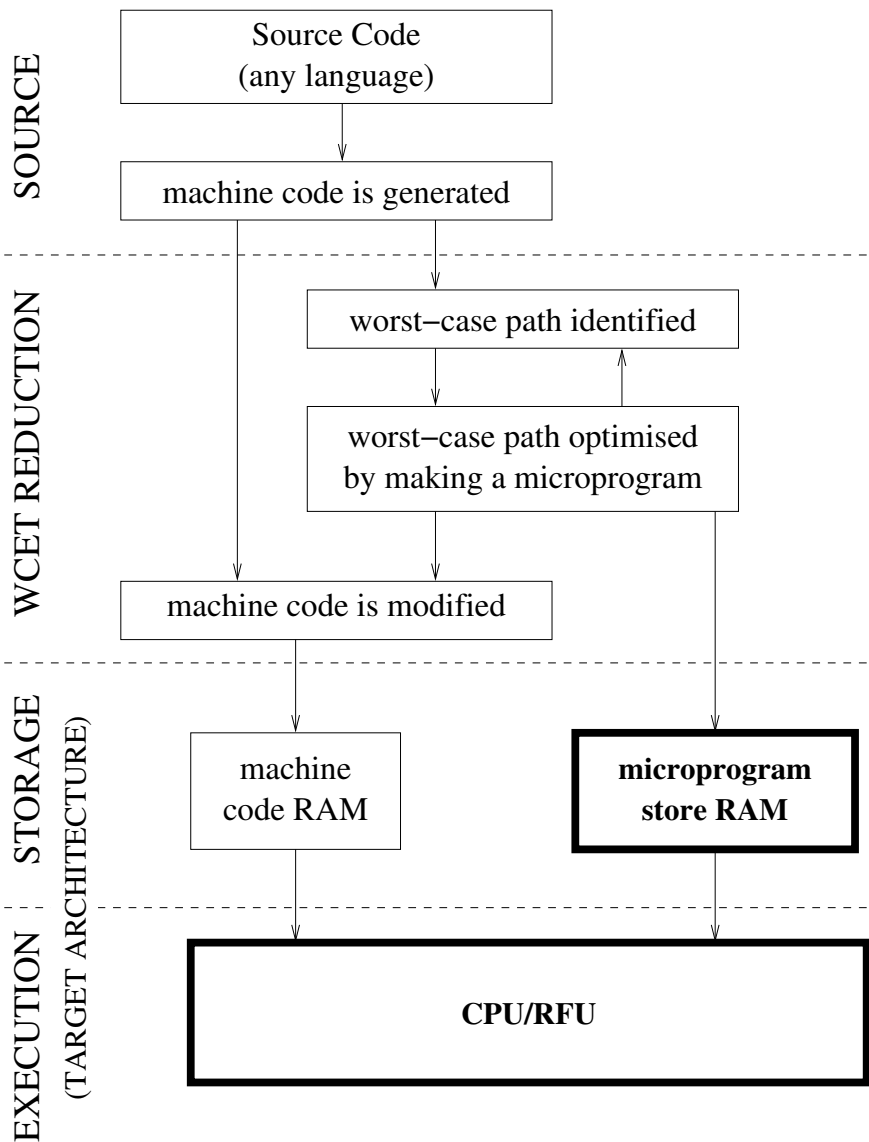


Figure 5.1: The components of the abstract WCET reduction process that are implemented in this chapter are highlighted in bold.

- A software program to convert machine code into microprograms for execution on the RFU;
- A cycle-accurate microarchitecture simulator.

Each of these components could be written from scratch in an appropriate HDL or software language. But this would force the CPU architecture parameters (such as l , m and n and the others listed in Table 4.1) to be fixed at design time. This would prevent exploration of the effects of different design parameters. Ideally, a generator is needed: a program that can produce an implementation of TARGET in some HDL, given some architecture parameters. This would be similar to the ASIP model used by [103, 18, 13], but it would include generation of an RFU and the associated software. The generator program would use a single machine specification to produce three compatible components (Figure 5.2).

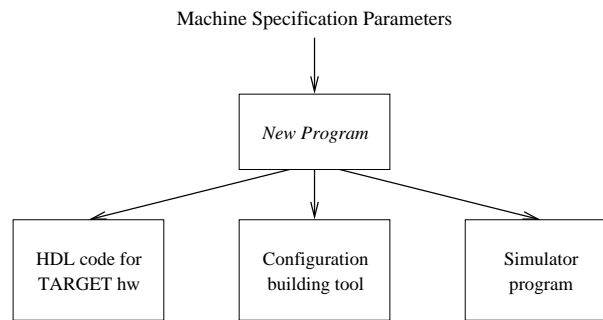


Figure 5.2: The components that are required for each implementation of TARGET are generated by a program, based on CPU architecture parameters. This allows experimentation with different parameters, and provides a way to assure compatibility between the components.

The general form of the required microarchitecture has been described in section 4.2, and previous work involving microprogram generation has been examined in section 2.6.8. However, the topic of simulation has not been adequately addressed. Therefore, section 5.1 examines some previous work in this area. This is followed by an overview of the first prototype implementation of the CPU generator for TARGET (section 5.2).

5.1 Simulation

Simulation is the process of modelling the behaviour of one system using another. Simulation allows experimentation with a new design while only parts of the design are complete, and facilitates automatic testing and evaluation because a simulated device can be manipulated by programs. A simulator is also useful for debugging a design because inspection and single-stepping are possible without invasive modifications to hardware.

Architectural simulators are programs that execute on a *host* system and provide a model of a CPU. They are often classed as either *functional simulators* or *performance simulators* [231]. Functional simulators are concerned with replicating the functionality of a CPU (the ISA) without concern for precisely replicating the methods used by the CPU (the *microarchitecture*). This simulates what a programmer would expect the CPU to do: all machine instructions produce the correct results. Performance simulators also do this, but by simulating the microarchitecture itself. This allows accurate timing information to be obtained and provides a more detailed simulation. But because of the higher level of detail, execution speed is reduced: a tradeoff has been made. Tradeoffs in simulator design are illustrated in Figure 5.3.

5.1.1 Existing Simulators

Existing simulators target many common types of CPU, often also simulating memory and some types of input/output device. The SimpleScalar [41] simulator suite, intended for industrial and research use, contains programs to simulate ARM processors and a MIPS-like CPU (PISA). These programs simulate processors at different levels of abstraction [41, 231], as shown in Table 5.2. More detailed simulators produce more accurate results, but are slower, allowing the user to choose an appropriate tradeoff from Figure 5.3.

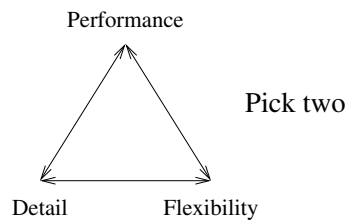


Figure 5.3: The tradeoffs in simulator design, from [231]. Performance is speed of simulation. Detail minimises risk from differences between the microarchitecture and the simulator. Flexibility allows experiments with related architectures. In general, it is not possible to optimise more than two of these factors.

Name	Speed	Detail	Description
sim-fast	Highest	Least	Functional, no checks.
sim-safe			Functional, with checks.
sim-uop	Slowest	Most	Functional, simulates part of microarchitecture.
sim-outorder			Performance, simulates entire microarchitecture.

Table 5.2: The different types of simulator in the SimpleScalar toolset. These simulators are sorted in order of timing accuracy: `sim-outorder` is the most accurate.

Simulators are extensively used for embedded development work. The OpenRISC development kit [150] includes a functional simulator for the OpenRISC processor, which also simulates some peripherals. The ARMulator is an official performance simulator from ARM Limited [14]. Bochs [153] is a functional x86 simulator. QEMU [32] provides functional simulation for several types of CPU including x86, ARM, SPARC, PowerPC and MIPS.

Simulators are also used for office tasks and entertainment. PowerPC versions of Virtual PC simulate an x86 processor with PC peripherals [12]. The Dosbox [65] simulator provides an DOS-era PC environment for “retro gaming”. These simulators are functional as high speed is essential for interactive use.

5.1.2 Simulator Validation

Architectural simulation carries a risk of introducing error through differences between the simulator implementation and the device being simulated.

Some errors are easily detectable at the functional level, as they produce incorrect results as instructions execute. Other implementation errors may be hidden by the architecture, manifesting themselves as timing errors. An erroneous implementation of a cache would fall into this category - the simulated device would produce correct results, but with incorrect timing behaviour.

Both types of error can be detected by comparison with a reference implementation. SimpleScalar was verified against ARMulator [232] and real hardware. The verification involved random testing by feeding random instructions and states as input to the simulator and the reference implementation (Figure 5.4). Performance testing of elements that cannot be modified externally (e.g. caches) is carried out in a similar way, but it may be necessary to use a sequence of instructions (such as a benchmark program) rather than a single instruction.

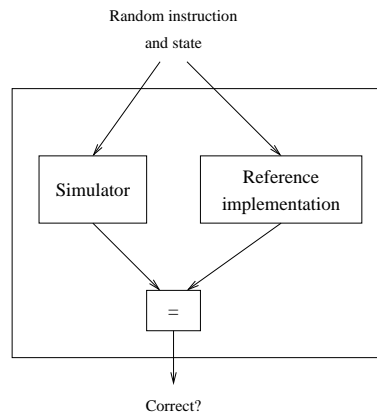


Figure 5.4: Verification of a simulator by random testing against a reference implementation, as used in the development of SimpleScalar [232].

5.1.3 Simulating an RFU

The TARGET RFU is composed of the same parts as the CPUs simulated by SimpleScalar, such as register files, RAM, and functional units, so the same simulation approach can be applied to TARGET. However, a performance simulator approach will be required, because the low-level timing properties of TARGET are important. Since a new CPU architecture is being designed, it will also be necessary to write the simulator from scratch.

5.2 MCGREP-1 TARGET Generator

The static implementation of the TARGET CPU generator is known as MCGREP-1, an acronym standing for **m**icroprogrammed **c**oarse **g**ained **r**econfigurable **p**rocessor. This generator produces one possible implementation of TARGET. Instances of TARGET are characterised by the parameters listed in Table 4.1, and since MCGREP-1 is a basic “proof of concept” implementation of the TARGET architecture, the feature set is deliberately limited in order to speed development. This means that full freedom to adjust the TARGET parameters is not available.

Section 5.2.1 describes the parameterisable features that are included in MCGREP-1, and section 5.2.2 describes non-parameterisable features. Subsequent parts of this section describe the microarchitecture, the RFU microprogramming software, and the simulator software. Section 5.3 has an overall evaluation of MCGREP-1.

The software language used to implement the MCGREP-1 generator and related tools is Python. Python [208] is used because it is a simple but powerful object-oriented language, ideal for rapidly implementing complex software for PCs and workstations. Python is not as useful as an embedded systems programming language since it is quite slow and has a large memory overhead, so embedded software for TARGET is implemented in C or assembly code.

5.2.1 Parameterisable Features

The MCGREP-1 generator implementation makes a tradeoff between configurability and speed of implementation. Because of the decision to aim for a basic proof of concept rather than a complete

tool, many of the configuration parameters are fixed. However, allowing flexibility in two areas actually reduces the difficulty of implementation. These two areas are examined below:

- **Microprogram Interface:**

Within the tools that generate an implementation of TARGET, there is a shared language. This is the microprogramming language that describes the operations of the RFU. It is used in the following places:

- Within the hardware microarchitecture (as described by HDL), where it describes the function of a decoder that translates microcode bits from RAM into control lines for the RFU components;
- Within the simulator, where it is used to interpret the meaning of microcode bits into actions to be taken by simulated components;
- Within the microprogram generator, where it is used to translate operations described at an encoding-independent higher level into low level microcode.

The language is different in each implementation of TARGET, because it is affected by all of the parameters. For example, if the set of ALU functions is extended, the microprogram language also needs to be extended to represent the new functions. It is therefore necessary to generate the language from the parameters.

Because this language is used throughout the MCGREP-1 software, and because it may be different in every instance of the CPU, a parameterisable *codec* was written first. Codecs are programs that encode and decode messages. In this case, the codec is a reversible mapping function, with the property that for any message m and any instance of the codec c , $m = c^{-1}(c(m))$. The encoded form $c(m)$ is microcode bits, and the decoded form m is a symbolic representation of actions to be taken. For example, the ALU command for addition is assigned the symbol `ARITH_ADD`, but the microcode for this function varies according to the parameters of the codec.

Instances of a codec class are initialised with a list of n symbols to be encoded. The number of bits used to encode these symbols is $\lceil \log_2 n \rceil$. (It is possible that logic usage could be reduced by careful encoding selection, but this is likely to be a minor optimisation, so the MCGREP-1 codec does not attempt to do this.) Each codec instance provides an `Encode` and a `Decode` method to translate to and from microcode bits.

The binary data produced by every codec is stored in a container object that represents the microprogram interface. This container is a memory manager for microcode bits. Figure 5.5 shows an example of the container in use. Briefly, the tasks of the container include:

- Allocating groups of microcode bits;
- Maintaining an index to allow each group of microcode bits to be referred to by name (e.g. `cmd` in Figure 5.5);
- Setting a group of microcode bits to a specific value;
- Obtaining the value of a group of microcode bits;
- Generating a RAM image of the microcode store for inclusion in a HDL source file;
- Obtaining microcode in binary form for storage and use within the simulator.


```

1 cmd_codec = Mapping_Function(ALL_COMMANDS)
2 container.Register(cmd_codec.Get_Code_Width(), 'cmd', None)
...
3 container.Set_Value('cmd', cmd_codec.Encode(op_type))
...
4 op_type = cmd_codec.Decode(container.Get_Value('cmd'))

```

Figure 5.5: Sample usage of the microcode container class and codec class. On line 1, a codec is created to represent the symbols in ALL_COMMANDS. On line 2, some bits of container are allocated to store a 'cmd', which controls some functional unit. As a microprogram is generated, an op_type is stored as a 'cmd' in the container (3). As the simulator executes this program, the op_type is recovered by decoding the 'cmd' field.

MCGREP-1 Microprogram Codec

▷ /mcgrepl/sw/mapping_function.py
Python implementation of parameterisable mapping function *c*.

▷ /mcgrepl/sw/ucode_controller.py
Python implementation of microprogram container class.

▷ /mcgrepl/sw/constant_table.py
Definitions of microprogramming symbols such as ARITH_ADD and ALL_COMMANDS.

▷ /mcgrepl/sw/c2.py
Make_Generic_Block_RAM: block RAM generator for Spartan-2E FPGAs.



These functions allow microcode to be generated, decoded for simulation or debugging, stored in hardware, or stored within a program. Like the codec class, the container class makes no assumptions about the type of microcode that will be used. It can support microcode of any width, with any number of encoded commands.

- **Microprogram Store Dimensions:**

Because the microprogram container class supports any number of bits, it is necessary to support a range of different memory configurations to store them. The memory that is used for microcode is “block RAM”: embedded memory found within Xilinx FPGAs. Each block RAM is a small two-port configurable RAM, supporting a range of different data and address bus widths [290]. Block RAMs are a space and energy efficient way to store microcode in an FPGA.

The microprogram container class is able to generate block RAMs of any size by cascading multiple block RAM units. It is also able to support two sizes of block RAM (used in different FPGA models), and can initialise the contents of the block RAM to a microprogram image.

5.2.2 Non-Parameterisable Features

The aim of maximising the simplicity of the MCGREP-1 generator leads to many parameters that cannot easily be changed, because assumptions about their values are made within the software. As stated earlier, there is a tradeoff between configurability and speed of implementation. The following parameters are fixed in all CPUs generated by MCGREP-1:

- **ISA and Built-in Microprogram:**

A microprogram is needed to operate each implementation of TARGET, executing software by cyclic execution (section 4.2.1). This acts in place of a hardwired (RISC-like) control unit. The RFU resources used to execute machine code are reused to execute custom microcode.

This microprogram defines the ISA of the CPU. Therefore, it is necessary to select an ISA before programming can begin. For MCGREP-1, the OpenRISC “ORBIS32” ISA [151, 150] was chosen because:

- Reusing an existing ISA enables reuse of compilers, linkers, OSs, code libraries, and other tools: saving the need to port these components from another ISA (section 4.2.1).
- The ORBIS32 ISA is not large. It contains 89 defined instructions [151], many of which are listed in Appendix C: OpenRISC ORBIS32 Quick Reference. 34 of these are never generated by the ORBIS32 version of `gcc`. In some cases, this is because they are special instructions which are used only by an OS, e.g. `l.rfe`: return from exception. In other cases, it is because the C compiler is not able to recognise when they will be useful, such as the `l.ff1` instruction shown in Figure 2.11. Eight of the instructions are also reserved for ASIP-style extensions.

Of the remaining 55 instructions, two are used for division. These are generated by the compiler, but they are optional as OpenRISC can be configured without a division unit to save space. If the compiler is executed with the `-msoft-div` option, a software division procedure is used instead.

The final 53 instructions are all used by compiled programs. However, the architecture design allows these instructions to be grouped by function, as shown in Table 5.3. Within each group, every instruction requires the same sequence of operations, with only a small difference in some control line settings. For example, all the “ALU (Immediate)” operations obtain source data in the same way, and write output in the same way. The only difference is in the setting of the ALU control input. This property allows a single microprogram to be written for every *group* rather than every *instruction*.

- The ORBIS32 ISA is already implemented by the simulator program `orlksim`. This enables functional verification of the MCGREP-1 simulator against execution traces emitted by `orlksim`.
- The ORBIS32 ISA is also implemented by the OpenRISC soft core, written in Verilog (section 2.5.1). Therefore, CPUs generated by MCGREP-1 can be implemented on the same FPGA as OpenRISC CPUs for direct comparison.
- The ORBIS32 ISA does not include any floating-point or vector instructions: the OpenRISC designers have defined these instructions as part of other instruction sets for OpenRISC. Omitting floating-point and vector instructions greatly simplifies implementation. Results from integer-only performance can be extrapolated to cover floating point

Group	No. of Insts.	Examples
ALU (Immediate)	5	<code>l.addi, l.muli</code>
ALU/Shift (Register)	9	<code>l.add, l.sll</code>
Compare (Immediate)	10	<code>l.sfeqi, l.sfgesi</code>
Compare (Register)	10	<code>l.sfeq, l.sfges</code>
Jump (Direct)	4	<code>l.jal</code>
Jump (Indirect)	2	<code>l.jr</code>
Load	5	<code>l.lbz, l.lws</code>
Move High	1	<code>l.movhi</code>
NOP	1	<code>l.nop</code>
Shift (Immediate)	3	<code>l.slli, l.srai</code>
Store	3	<code>l.sw, l.sh</code>

Table 5.3: Functional groups in the OpenRISC ORBIS32 ISA (see also Appendix C).

and vector operations since these types of instruction are just extended forms of integer ALU instructions.

Other RISC ISAs exist, with compiler tools, a small set of function groups, and simulators. However, soft cores that implement these ISAs are unsuitable for a variety of reasons:

- **MIPS:** the Opencores website [189] has several CPUs that implement part of the MIPS ISA, but these are incomplete unofficial MIPS clones, and not comparable to a real MIPS CPU.
- **Microblaze** [286]: this currently lacks a freely available ISA simulator.
- **LEON-2/3** [97]: HDL source, compiler tools and a simulator are available for this implementation of the SPARC RISC ISA. However, the SPARC V8 architecture is considered to be too complex, with powerful features such as windowed register files requiring extra engineering effort [237]. Choosing SPARC would add significantly to development time over the simpler ORBIS32 ISA.

In MCGREP-1, the ORBIS32 ISA implementation is generated by a software method that registers a handler for each instruction. The registration method (Figure 5.6(a)) specifies both the opcode format and a method that will generate the microprogram. This method is the same for all members of a group (Table 5.3), differing only in control line setup. An example for the “ALU (Register)” group is shown in Figure 5.6(b).

From this, the MCGREP-1 software generates:

- A microprogram to support all of the instructions that have been declared;
- A dispatch unit that converts an instruction word into a microprogram address - the address of the function to handle that instruction.

The microprogram also contains some special programs as listed in Table 5.4. These are not instruction handlers: rather, they are housekeeping functions for the CPU/RFU.

- **Interconnect** and l, m, n :

```

(a) Opcode(UCode_ALU, [L_ADD], ALWAYS, 1, 0x38, [(0x0f, 0x00)])
...
(b) def UCode_ALU(seq, alloc, x):
    if ( x == L_ADD ):
        simple = ARITH_ADD
    ...
    alloc(REG_FIELD_1, REG_FIELD_2, REG_FIELD_0,
          simple, None)

```

Figure 5.6: A part of the ORBIS32 ISA implementation for MCGREP-1. The code in (a) specifies the opcode for register-register addition: the top six bits of this opcode (1.add, Appendix C) are 0x38, and when the bitmask 0xf is applied to the instruction word, the result is 0. The code in (b) describes the microcode for every member of the “ALU (Register)” group (Table 5.3) including register-register addition.

Name	Purpose
Boot	Initialises the CPU.
IllegalEx	Handles illegal instructions by jumping to address 0x700 as specified by the OpenRISC manual [151].
AlignmentEx	Handles memory alignment errors [87] by jumping to address 0x600.

Table 5.4: Special microprograms for CPUs generated by MCGREP-1.

Architectures generated by MCGREP-1 are totally connected RFUs, i.e. every register file port is accessible from every functional unit port. This is possible because the size of the RFU is limited to two elements. Referring to Figure 4.5, $l = 1$, $m = 0$, and $n = 1$. This means that there is one CPU-capable element (n), and one element that contains only an ALU (l).

This sort of architecture is very restrictive, but it should be sufficient to demonstrate *some* speed increase through the parallelism provided by two RFU elements rather than one.

- **ALU function set:**

The ALU function set generated by MCGREP-1 is restricted to the functions required to implement the OpenRISC ORBIS32 ISA without division functionality.

- **Register file dimensions:**

The register files generated by MCGREP-1 are restricted by the dimensions of the FPGA block RAM used to implement them. Xilinx Spartan-2E and later FPGAs can provide memory resources for at least 256 registers of size 32 bits by using two block RAMs connected as a pair. 31 of these are used as *general-purpose registers* (GPRs) for OpenRISC programs ($r0$ is always zero). A number of additional registers are used to store system variables such as the program counter. The rest are available for use for any purpose by microcode.

Block RAM is an extremely effective way to implement any memory on an FPGA, including register files, scratchpads and microprogram stores. Some FPGA-based CPU designs implement register files using a 2D matrix of flip-flops: Microblaze [286] is an example, and OpenRISC can be configured in this way. The advantage of such implementations is

MCGREP-1 Non-parameterisable Components

▷ /mcgprep1/sw/ucode_utilities.py
 UCode_ALU: this method generates the microprogram to implement an ALU operation such as addition.

▷ /mcgprep1/sw/c2.py
 Make_Decoder: this method builds the ORBIS32 ISA interpreter (microprogram and instruction decoder).

▷ /mcgprep1/sw/c2.py
 Make_Exec_Unit: this method creates VHDL code to implement each ALU.

▷ /mcgprep1/sw/codegen.py
 Make_UOP_Function: this method creates Python code to simulate each ALU.



Parameter	Configurable?	Notes
ALU function set	no	
Interconnect	no	
ISA	no	
l, m, n values	no	
Register file dimensions	no	
Built-in microprogram	no	
Microcode interface	yes	Automatically generated
μ code store dimensions	yes	Resizable

Table 5.5: Parameters for initial generator implementation.

that more register file access ports are available, but the cost is that a greater FPGA area is consumed.

5.2.3 Microarchitecture

The complete list of TARGET parameters represented in MCGREP-1 is shown in Table 5.5. This section describes the implementation of the microarchitecture generated by MCGREP-1.

In MCGREP-1, VHDL [17] templates are used to produce HDL code. These templates define the majority of the functionality, but computed data is substituted into the HDL as necessary. This is done both for fully parameterised components such as the microprogram interface (section 5.2.1) and for non-parameterised components such as the ISA interpreter (section 5.2.2) which are partly generated by software. A block diagram of every MCGREP-1 CPU is shown in Figure 5.7. The parts of Figure 5.7 are (from left to right):

1. LSU:

The load/store unit (LSU) is a standard CPU component, acting as an interface between memory and CPU internals. The MCGREP-1 LSU can load or store 8, 16 or 32 bit data in

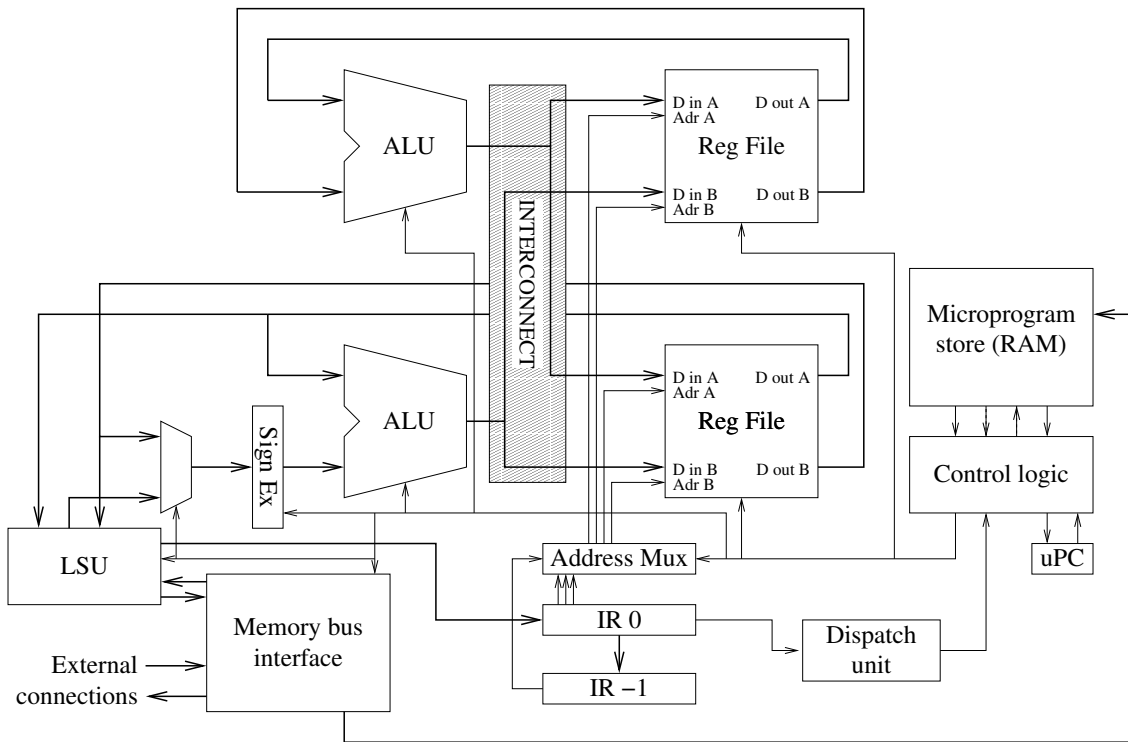


Figure 5.7: Architecture diagram for every version of MCGREP-1. The configurable features of the hardware are found within the control logic.

a 32 bit address space. It is a simple LSU, and does not cache information or attempt to prefetch instructions.

The only feature that sets the MCGREP-1 LSU apart from other simple LSUs is that it only has partial support for *byte steering* [286]. Wide buses are divided into byte lanes. At the bus level, memory stored in one byte lane can only be read back on that byte lane. However, byte-wide load and store instructions always use the least significant byte regardless of the memory address, so read and write byte steering is performed by the LSU to ensure that data is marshaled appropriately for the address in use. In MCGREP-1, this marshaling is handled by the microprogram. Read steering is entirely carried out using the shift feature of the ALU. Write steering is partly carried out by the LSU under the direction of the microprogram. This simplifies the LSU at the cost of adding complexity to the microprogram.

2. Sign Extender:

A *sign extension* unit is a requirement for implementation of the ORBIS32 ISA. Sign extension is the replacement of the x most significant bits of a word with the value of the most significant bit. This is used for conversion of 8 and 16 bit integers to 32 bit values, as it preserves the correct value of negative numbers. OpenRISC uses sign-extended addition as part of many operations including branch, load and store.

The MCGREP-1 sign extension unit provides the four types of sign extension required by ORBIS32 (Table 5.6). It also provides a pass-through mode, and can sign extend 8 and 24 bit values in order to support the LSU's read steering requirements.

MCGREP-1 Symbol	Action
SIGN_EXTEND_16	Sign extend bottom 16 bits
SIGN_EXTEND_JUMP	Sign extend bottom 26 bits
ZERO_EXTEND_16	Top 16 bits replaced with zero
SIGN_EXTEND_SPECIAL	$out[31:16] = in[25]$ $out[15:11] = in[25:21]$ $out[10:0] = in[10:0]$

Table 5.6: Types of sign extension required by OpenRISC ORBIS32 ISA.

MCGREP-1 Symbol	Action
ARITH_ADD	$out = inA + inB$
ARITH_MUL	$out = inA * inB$
ARITH_SUBTRACT	$out = inA - inB$
LOGIC_AND	$out = inA \& inB$
LOGIC_OR	$out = inA inB$
LOGIC_XOR	$out = inA \text{ xor } inB$
PASS	$out = inB$
SHIFT_LEFT	$out = inB \ll inA$
SHIFT_RIGHT_ARITHMETIC	$out = inB \gg inA$ then sign extend out
SHIFT_RIGHT	$out = inB \gg inA$

Table 5.7: Types of ALU operation required by OpenRISC ORBIS32 ISA.

3. Memory Bus Interface:

The MCGREP-1 memory bus interface connects the internal memory bus used by the LSU to an external Wishbone [190] bus, a standard bus for CPU soft cores which enables connection to a variety of co-processor devices from Opencores [189] and elsewhere. The only unusual feature of this bus interface unit is the inclusion of a special reserved memory area representing the microprogram store. Writes to this special memory area go directly into microprogram memory, allowing programs to define new microinstructions.

4. ALUs:

Both of the ALUs in Figure 5.7 have identical functionality. The functions are limited to those needed to support the OpenRISC ORBIS32 ISA (Table 5.7). The HDL specifies these functions using a VHDL `case` statement, making the (possibly optimistic) assumption that the synthesis tool will be able to simplify the ALU. A similar strategy is used within the OpenRISC soft core.

The ALU also carries out comparison operations, since these are partly an extension of the subtraction function. The comparison unit is similar to the OpenRISC implementation, but is able to take the code specifying the *type* of comparison from either the instruction register or the microprogram store. There are ten types of comparison [151], so a four bit code is used. The ability to obtain the code from two sources allows microprograms to carry out the same set of comparison operations as OpenRISC code.

An alternative implementation would encode the comparison type in the microprogram. This

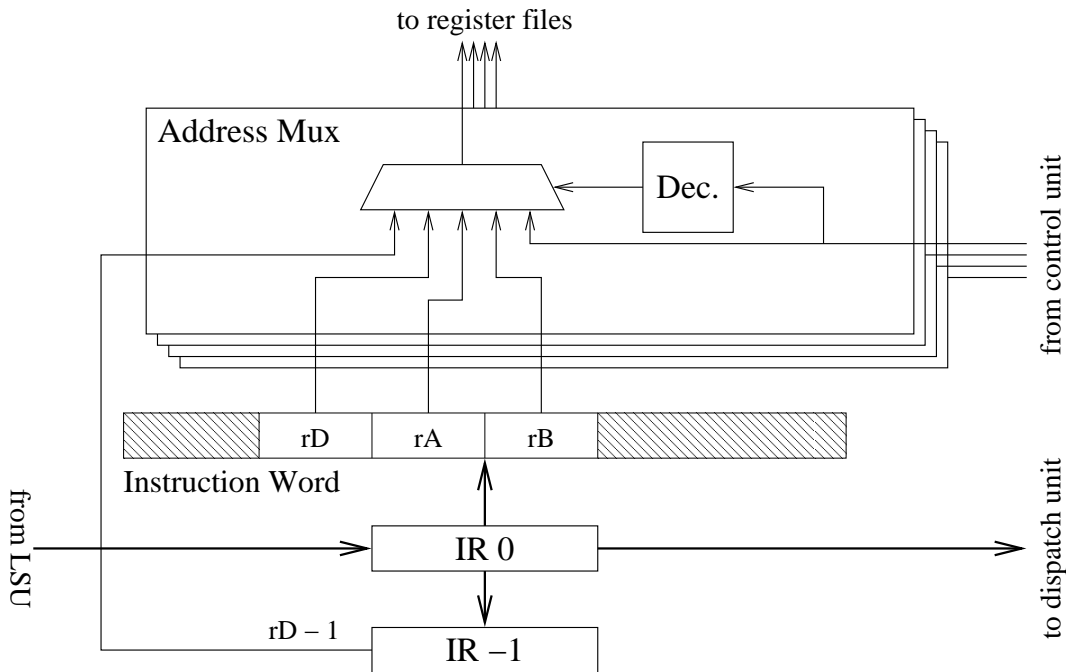


Figure 5.8: Address multiplexer, in detail. The multiplexer allows the addresses of the four register file ports to be set from the register fields of the instruction word, or directly from the microprogram.

would require a different microprogram for each type of comparison, taking up valuable space in the microprogram store. As in the case of the LSU's byte steering system, there is a tradeoff here between microprogram space and logic usage. The cost of an extra multiplexer appears to be lower than 10-20 extra microinstructions, so MCGREP-1 takes this approach.

5. Interconnect:

MCGREP-1 opts for a fully connected RFU. Since only two register files are involved, the interconnection can also assure that both register files always have the same contents. This is done by distributing all writes to every register file. Unfortunately, this scheme cannot scale to support more register files and more ALUs. More scalable schemes will be needed for larger interconnects, as previously discussed in section 4.2.3.

6. Address Mux and Instruction Registers:

The address *multiplexer* (mux) device controls the address ports of the register files, using information from the current instruction register (IR 0) and the previous instruction register (IR -1). Figure 5.8 shows a detailed view of these. The purpose of the address mux device is to allow the address of each register file port to be set up for any of the following three scenarios:

- (a) Fetching register values, before instruction execution. In this scenario, the register ports need to be set according to the rA and rB fields of instruction register IR 0.
- (b) Updating a register, after instruction execution. In this scenario, the register ports need to be set according to the rD field of instruction register IR 0. In some cases, the previous value of the rD field is used from IR -1. One case where this is necessary is during store

execution, because the operation of fetching the next instruction must complete before the memory access begins.

- (c) Executing microprograms. In this scenario, the microprogram must have full control of all register ports. Neither instruction register is used.

Support for these three use cases is sufficient to support ORBIS32 execution, and also gives the microprogram full flexibility to use the register files as required.

7. Register Files:

The MCGREP-1 register files are implemented using block RAM. This restricts the dimensions of each RAM, but also restricts the number of ports available. Each block RAM is a dual-ported memory [290]. The ports are independent, and each can be used in any clock cycle for reading or writing. However, this means that:

- At most two registers can be read from a block RAM in each clock cycle; or
- At most two registers can be written; or
- At most one register can be read while at most one register is written.

These restrictions would seem to prevent one OpenRISC instruction being executed every cycle, if block RAM is used for the register file, because many OpenRISC instructions use two inputs and produce one output. This is two reads and one write. The OpenRISC soft core avoids this problem using one of two strategies, which are selectable using a configuration file [150]:

- The register file is not implemented using block RAM: instead, a three-port memory is created on FPGA logic. This strategy, which is also used by Microblaze [286], increases the FPGA logic cost of the CPU.
- Two block RAMs are used in place of one. On each block RAM, one port is dedicated for writes and the other is dedicated for reads. All block RAM writes are sent to both memories at the same time.

However, the register access restriction is not as important on MCGREP-1 as it is on OpenRISC, because CPUs generated by MCGREP-1 always operate by cyclic execution (section 4.2.1). Taking two clock cycles to execute an arithmetic operation is not a problem because two cycles are required to fetch the next instruction. Therefore, the same ports can be shared between reads and writes.

8. Dispatch Unit:

The dispatch unit takes a machine instruction as input and outputs a microprogram address. It is used by the control hardware to decode instructions (Table 5.4).

In MCGREP-1, the dispatch unit is generated from the ISA description (section 5.2.2). The scheme used for this is a decoding tree. During the declaration of each instruction (Figure 5.6), the unique features of the machine code representation of that instruction are declared. These are placed into an appropriate position in a tree in which every non-leaf node represents a decoding step, as shown in Figure 5.9.

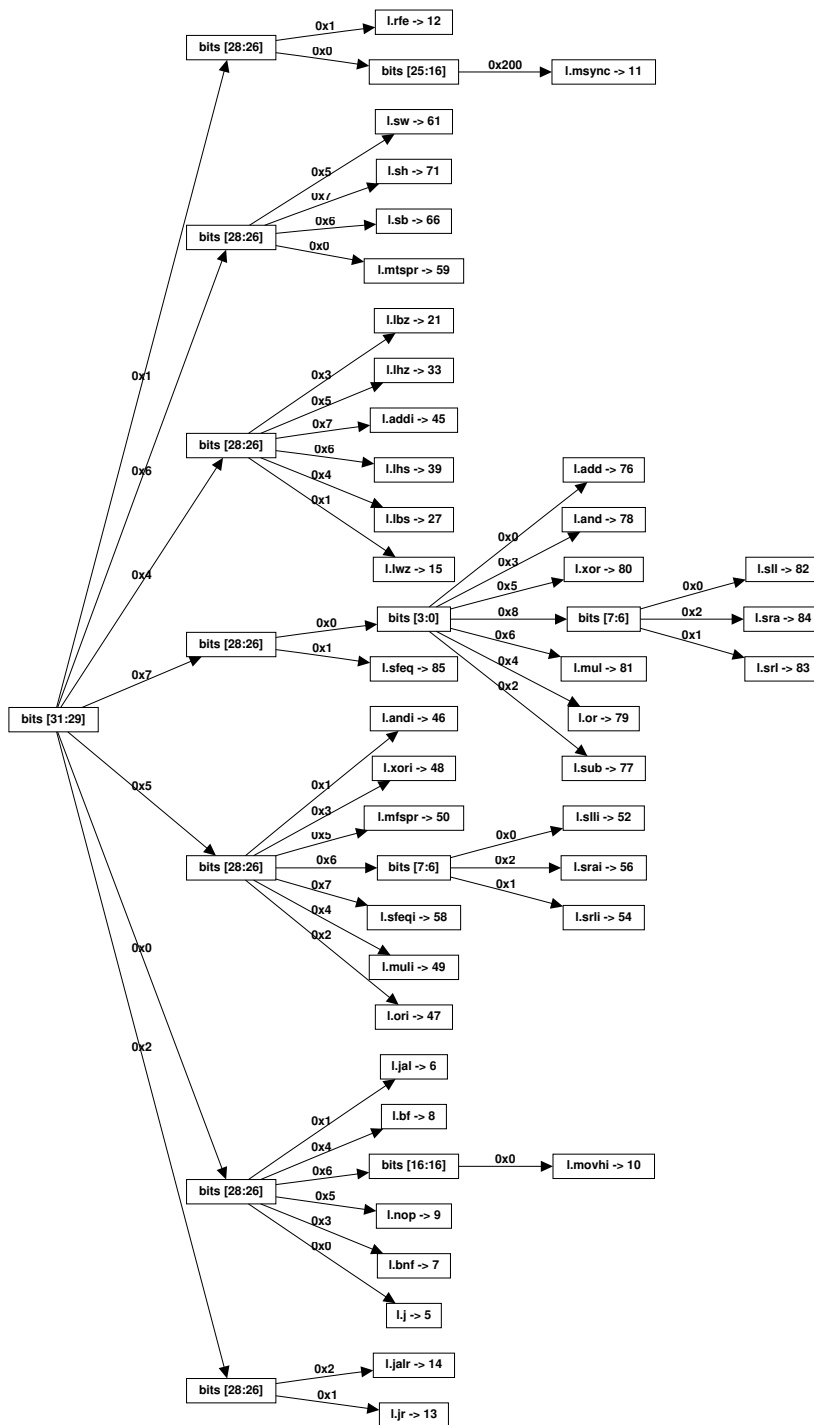


Figure 5.9: Decoding tree for ORBIS32 as used in MCGREP-1. This tree is generated automatically from the ISA description, and then used to produce a hardware device that translates a machine instruction into a microprogram address, described by nested `case` statements in VHDL. In this tree, leaf nodes represent instructions and their associated addresses. Non-leaf nodes represent tests applied to the bits of the machine instructions. See also: Appendix C, OpenRISC ORBIS32 Quick Reference.

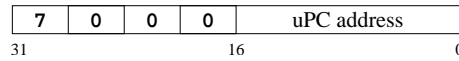


Figure 5.10: The MCGREP-1 branch to microprogram instruction format. The low 16 bits specify the target microprogram address. In the ORBIS32 ISA, branch to microprogram instructions occupy space reserved for application-specific custom instructions.

Symbol	Function
UCTRL_END_OF_INSTRUCTION	End of instruction: set uPC to output of dispatch unit
UCTRL_END_OF_JUMP_INSTRUCTION	As above, but also inhibit interrupts
UCTRL_JR_EA	Relative uPC jump based on low two bits of effective address (this is used for load and store operations)
UCTRL_JR_K	Relative uPC jump
UCTRL_JP_A	Absolute uPC jump
UCTRL_JP_A_IF_ACTIVE_ELSE_JR_1	Conditional absolute jump

Table 5.8: uPC branch commands supported by MCGREP-1. Not shown: some of these commands also have a “WHEN READY” form that inhibits branching until the LSU has completed a transaction.

This is an efficient and flexible way to translate instructions into microprogram addresses. It is efficient because a hardware version of the decoder carries out all decoding steps simultaneously, and flexible because the tree is generated from any number of simple instruction declarations.

Figure 5.9 does not illustrate the handling of illegal (unrecognised) instructions, which always map to a specific handler microprogram (Table 5.4). It also omits the special MCGREP-1 “branch to microprogram” instruction, which begins execution of a custom microprogram. This instruction is an extension to the ORBIS32 ISA, with the format shown in Figure 5.10.

9. Microprogram Store:

The microprogram store produced by MCGREP-1 is composed of a number of block RAMs. This component is generated automatically as described in section 5.2.1.

10. Control Logic and uPC:

The control logic is partly composed of a microprogram decoder, mapping microprogram bit patterns to MCGREP-1 symbols as described in section 5.2.1. Another part manages the *microprogram counter* (uPC), a register containing the address of the next microprogram line to be executed. Microprograms can affect the setting of uPC in a variety of ways (Table 5.8). The uPC commands are intended to support both the ORBIS32 ISA and custom microcode.

5.2.4 Microcode Generator

The preceding sections describe the MCGREP-1 generator and the microarchitecture it produces. But as noted at the beginning of this chapter, the microarchitecture is only one part of MCGREP-1. A microcode generator is needed to make use of the RFU, which is controlled by a microprogram.

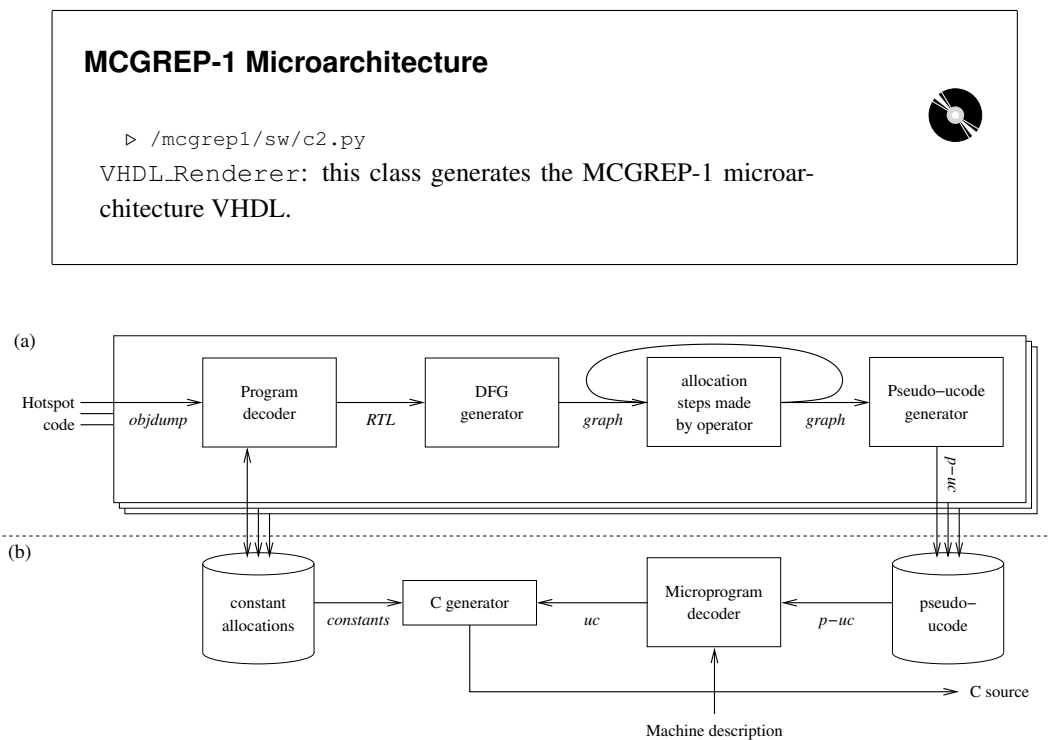


Figure 5.11: Diagram of the microcode generator for MCGREP-1. Two major steps are involved. (a) is applied for each WC path fragment (a block of code that contributes most significantly to the WC path through the program). This step uses machine code in the form of a disassembly trace (from `objdump`). (b) is then applied once to generate microcode for a specific version of MCGREP-1. The output is C source containing embedded microcode, which implements sequences of RFU configurations.

The goal of the microcode generator tool is to reduce the execution time of the software fragments that contribute most significantly to the WC path through the program (section 4.3.5). It translates machine code into sequences of RFU configurations - a microprogram - and embeds the resulting microcode in C source, enabling WC path optimisations. At this stage, the goal is not to apply WCET reductions automatically as illustrated in Figure 5.1. Manual programmer-directed WCET reductions are sufficient to demonstrate the capabilities of MCGREP-1.

Some approaches for microcode generation have already been discussed in section 2.6.8: trace scheduling is one possibility [85]. But because MCGREP-1 is a proof of concept, a partly manual implementation approach is used at this stage. However, unlike the manual approaches used to configure some CGRAs, the microcode generator tool provides a *graphical user interface* (GUI) to assist with the process, and enforces rules that preserve functional correctness.

The steps carried out by the MCGREP-1 microcode generator are shown in Figure 5.11. In line with the aim of using the RFU to exploit ILP in software, the microcode generator tool uses ORBIS32 machine code as its input. The GUI permits a human operator to rearrange the microoperations that make up the machine code so that ILP is exploited. In effect, this is manual microprogram compaction. It does not scale well, but serves as a demonstration of what *could* be achieved by an automatic tool if one was written. An annotated screenshot of the rearrangement GUI is shown in Figure 5.12.

The microcode generator tool uses a symbolic code that is independent of some of the RFU pa-

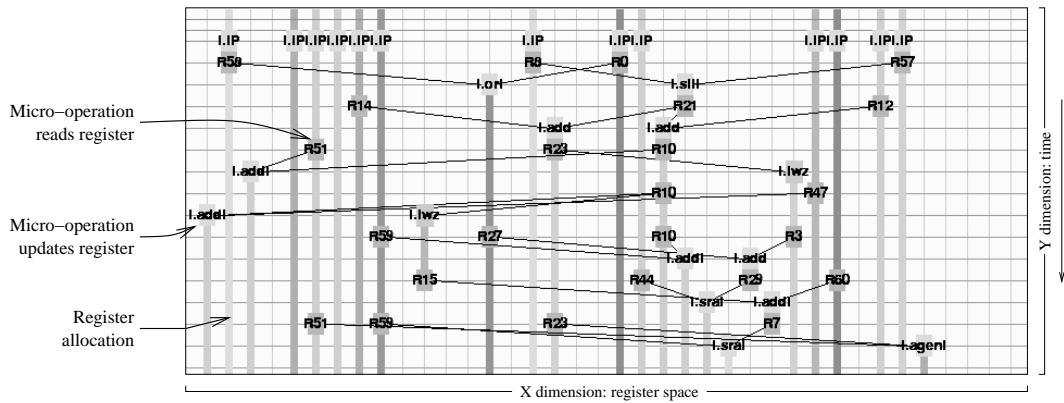


Figure 5.12: GUI for micro-operation scheduling in MCGREP-1. The GUI allows register resources and ALUs to be rearranged to minimise execution time. Micro-operations can be moved horizontally and vertically using a mouse in order to achieve this.

MCGREP-1 Microprogram Generator

▷ /mcgprep1/testcases/build

Builds all testcases, using both gcc and the microprogram generator.

▷ /mcgprep1/sw/aid_rtl_main.py

Code_Generator: this method generates microcode for a WC path fragment using the pseudo-microcode database. It may start the GUI if no database entry exists yet for that path.



rameters. This is stored in a database (on disk), so that multiple optimisations can be independently applied and adjusted within a program. The database provides communications between parts (a) and (b) of the process in Figure 5.11.

The pseudo-microcode is converted into microprogram data using the interfaces described in section 5.2.1. This is then transformed into C code, using a structure like the one shown in Figure 2.22. An “Upload” procedure is also generated: when called by the embedded software, this copies data from a microprogram structure into the microprogram RAM. Once an upload has taken place, the branch to microprogram instruction (Figure 5.10) can be used to start the execution of custom microcode.

However, there is another type of data that must be uploaded. Conventional ORBIS32 instructions make heavy use of *immediate values*: constants that are embedded in the machine code. At runtime, the register file provides storage for these, so the constants need to be allocated to distinct slots in the register file without conflicts between different microprograms. This is the reason for the inclusion of a second database for constant allocations. Constant data is uploaded by generated C code at the same time as the microprogram upload.

5.2.5 Simulator

A cycle-accurate microarchitecture simulator is a desirable part of the MCGREP-1 tools (Figure 5.2). As well as permitting experiments to be carried out without FPGA hardware, a simulator simplifies debugging, functional testing and evaluation. It can also be used to obtain the basic block timing measurements needed for WCET analysis.

The MCGREP-1 simulator is a performance simulator (section 5.1), and models the entire microarchitecture at the functional unit level. As testing of the simulator reveals in section 5.3.1, timing is identical to hardware implementations generated by MCGREP-1, given identical memory latency. The RFU is simulated, so the simulator executes both OpenRISC programs and custom microcode. It is possible to upload new microcode at runtime and have it execute immediately. Two other simple devices are also simulated:

- A RAM (of configurable size) is simulated. This is initialised with a binary image file, containing a program to be executed. The memory latency is configurable.
- A simple serial port is simulated. Data written to this serial port is written to the simulator console.

The simulator is a microprogram interpreter. It executes binary code from the microprogram memory, which is initialised with the built-in microprogram (section 5.2.2). The interface discussed in section 5.2.1 is used to translate binary microprogram data into symbolic form, and then the symbolic representation is interpreted by software methods that simulate each microarchitecture component in Figure 5.7. To maximise ease of implementation, the MCGREP-1 simulator was written entirely in Python [208], the language used to implement the other parts of the MCGREP-1 generator.

5.3 MCGREP-1 Evaluation

MCGREP-1 aims to generate a subset of possible implementations of TARGET, allowing these to be evaluated directly with fewer assumptions than were used for the theoretical analysis in section 4.3. The implementation fits into the abstract WCET reduction process of chapter 4 as shown in Figure 5.13, and needs to be evaluated in three distinct ways:

- For correctness: do the simulator and hardware execute programs and custom microcode correctly? (section 5.3.1); and
- Against the TARGET parameters: what subset of TARGET does MCGREP-1 implement? (section 5.3.2); and
- Against the requirements: how well does MCGREP-1 meet the requirements for the work? (section 5.3.3).

With the exception of Microblaze-based designs, the programs and hardware designs used to carry out these evaluations are part of Appendix A. Build instructions can be found in sections A.4 through A.6.

An important feature of TARGET is the ability to run any code. Benchmark programs provide a good way to test this as well as providing a way to check that the WCETs of real programs can be reduced. However, benchmark selection is a difficult problem, because the programs need

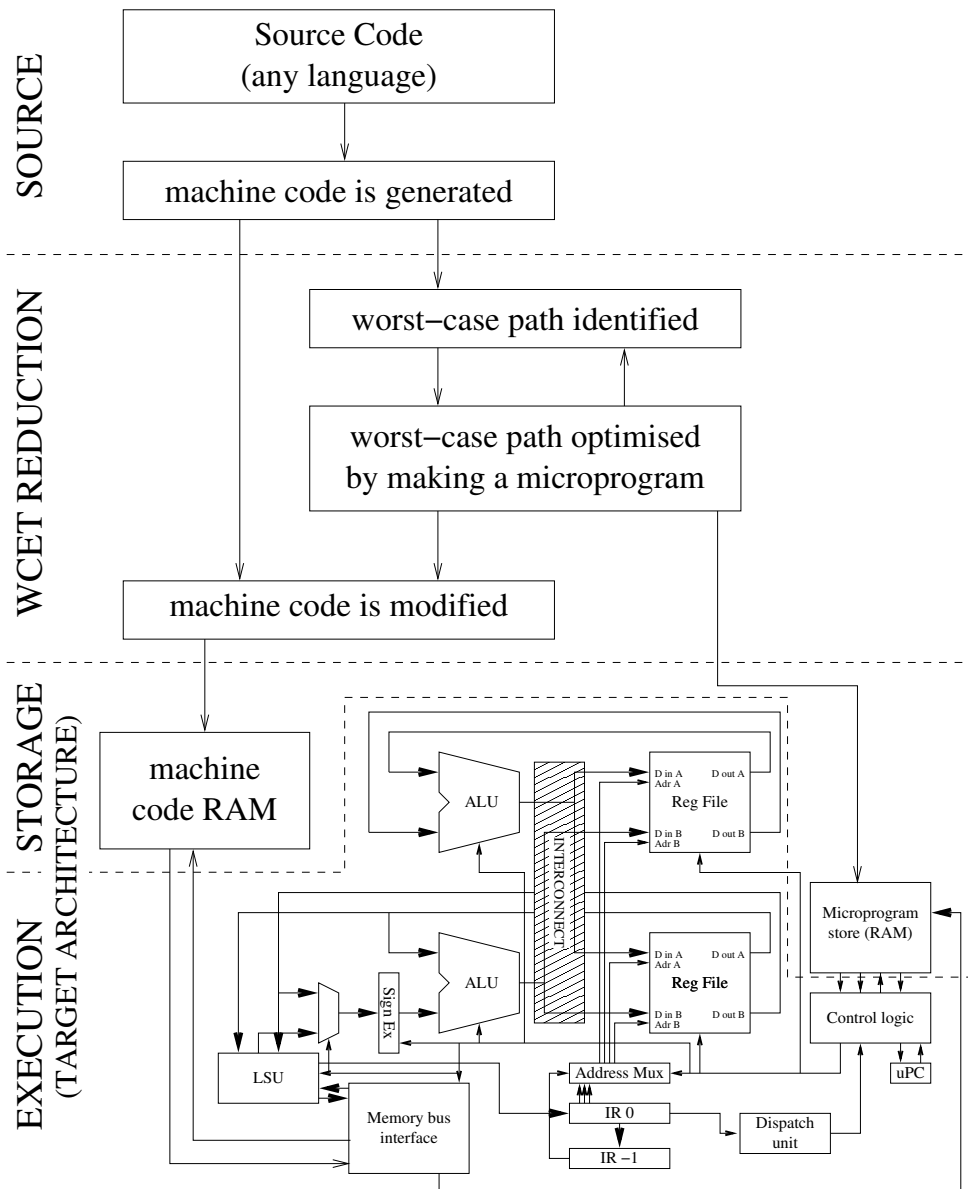


Figure 5.13: The four conceptual levels of the WCET reduction process using MCGREP-1.

MCGREP-1 Simulator

```
▷ /mcgrep1/sw/c2.py
```

Regular_Execution: this method sets up and starts a simulation.

```
▷ /mcgrep1/testcases/crc32/run_orig.py
```

Starts the simulator for the `crc32` test case, with custom microprograms disabled.

```
▷ /mcgrep1/testcases/crc32/run_accel.py
```

Same as `run_orig.py`, but custom microprograms are enabled, reducing execution time.



to be representative of a real workload. For this reason, programs from the MIBench [108] and Mediabench [154] libraries are used. These benchmark libraries are intended to be representative of embedded systems code. They include code often executed by embedded systems, such as media decompression, encryption and network algorithms. The well known SPEC benchmarks would be less suitable since they are intended to be representative of all code and therefore include many applications that would not be found on embedded systems [75]. Being embedded systems benchmarks, the EEMBC benchmarks [70] would also be suitable, but they could not be included in Appendix A as they are not freely redistributable.

In principle, any of the MIBench/Mediabench programs could be used, but a subset has to be chosen because of hardware limitations. Firstly, some programs require more memory than is available on a “BurchEd B5” Spartan-2E FPGA prototyping board, which provides only 256Kb of SRAM. This means that benchmark programs which operate on large data sets are not practical, because there is insufficient storage for the input. Secondly, benchmarks requiring an operating system and/or filesystem access are not suitable unless the accesses can be rerouted into memory. This is not always possible, particularly since it is common practice to load the contents of a file into memory before processing. When the filesystem is also in memory, this doubles the overall memory requirements. Thirdly, some benchmark programs require hardware floating point support, which is not currently implemented by MCGREP-1 CPUs.

Therefore, a subset of the available programs is used for evaluation (Table 5.9). Attempts were made to maximise the size of this subset without biasing the selection process to specific types of benchmark. For example, modifications to reroute data accesses to a memory filesystem were performed for `jpeg`, `dijkstra` and `mad` so that these programs could be included. However, other MIBench/Mediabench programs would not fit within the available resources, so they had to be excluded. It is likely that the subset is not representative of the real tasks that might be assigned to an embedded system, but this is not a problematic issue in this work because the benchmark code has only one purpose: verifying that MCGREP-1 CPUs operate correctly. It is most important that the benchmark code should exercise the software in order to identify bugs.

To this end, the chosen subset includes a range of programs with diverse types of hotspot, ranging from very large sequences of code (in `aes`) to small loops (`crc32`, `sha`), and with various amounts of control flow within hotspots: `dijkstra`, for example, has many conditionals within its hotspot while `crc32` has only one loop. Together, the benchmarks exercise nearly all of the machine code instructions in the implemented subset of ORBIS32. However, some microcode is not used, so an additional `coverage_test` program was added to the corpus. This test includes uncommon instructions such as `l.muli`, and tests the effects of illegal instruction and alignment exceptions.

5.3.1 Evaluation for Correctness

Functional testing ensures that a program operates correctly, i.e. that the correct results are produced for a given input. Three types of test are carried out, each using all of the benchmark programs in Table 5.9:

- **Machine code execution.** Because the MCGREP-1 ISA is OpenRISC ORBIS32, it is possible to validate the operation of MCGREP-1 CPUs against the OpenRISC simulator [150] (Figure 5.14(a)). This ensures that machine code is interpreted in the same way by both CPUs.

Benchmark	Purpose
aes	Encryption algorithm
crc32	Checksum computation
dijkstra	Shortest-path algorithm
g721	Telephone-quality voice compression
jpeg	Image decompressor
mad	MPEG audio decompressor
qsort	Sort algorithm
sha	Cryptographic hash algorithm
coverage_test	Tests all microprogram states

Table 5.9: Benchmarks used as a test corpus for MCGREP-1 CPUs.

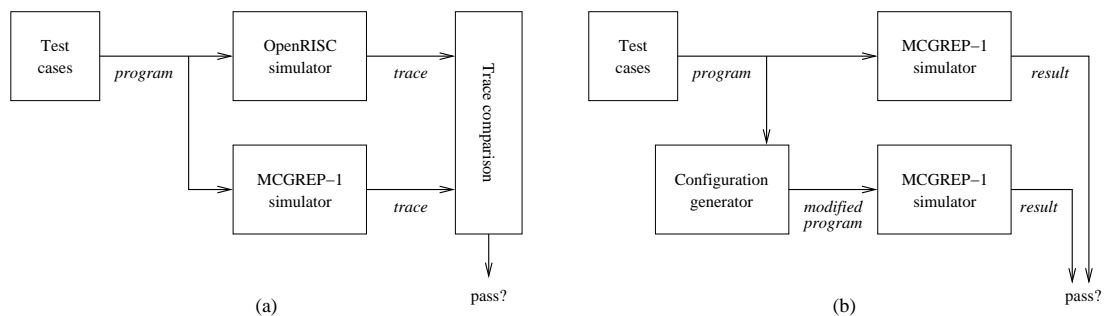


Figure 5.14: Diagram of the testing arrangements for MCGREP-1. In arrangement (a), execution of a program on a MCGREP-1 CPU is compared at the instruction trace level with execution of the same program on the reference (unmodified) OpenRISC simulator [150]. In arrangement (b), execution of a pure machine code program is compared with execution of a modified program that uses custom microcode, since microcode is intended to be an equivalent replacement for machine code.

- **Microcode versus Machine code.** Because the microcode generator produces microprograms that are equivalent to ORBIS32 machine code, it is possible to validate the microcode generator by comparing the results of “enhanced” programs with original code (Figure 5.14(b)).
- **Hardware versus Simulator.** The hardware can be compared against simulation to ensure that both behave in the same way.

For the final type of test, an FPGA-based debugging monitor was built to provide a user interface for an MCGREP-1 CPU. The architecture of this monitor is shown in Figure 5.15: it includes a T80 soft CPU core [269], which is used as a microcontroller, and a set of custom devices that interact with MCGREP-1 control lines. The states of internal registers and control lines are displayed on screen (Figure 5.16), and the CPU core can be single-stepped using keyboard commands. This manual process was used to compare operation of the simulator and the hardware. The CPU was tested on a “BurchEd B5” Spartan-2E prototyping board with 256Kb of RAM.

The results of all three types of test showed that the simulator and hardware generated by MCGREP-1 operated correctly when executing each of the benchmarks listed in Table 5.9. Repetitions of these tests using the RFU to reduce the execution time also produced the correct results.

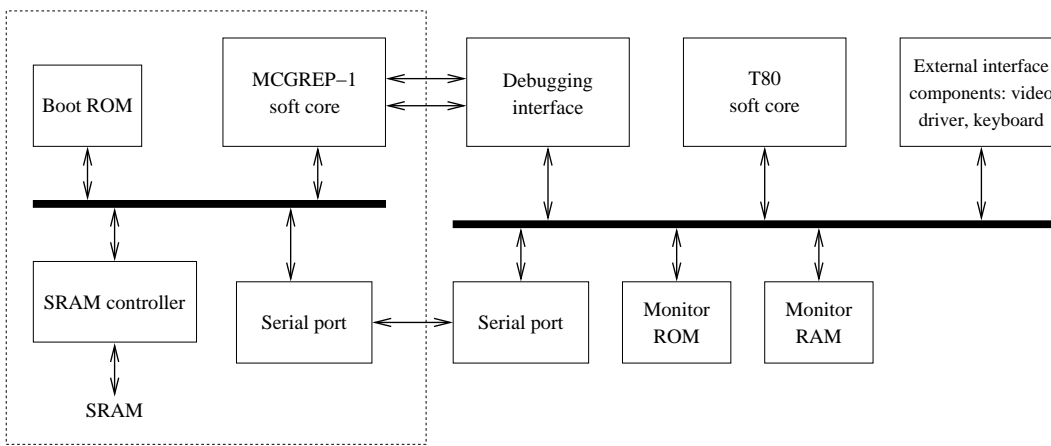


Figure 5.15: Architecture of the MCGREP-1 debugging monitor. The debugging monitor is an embedded system that monitors the MCGREP-1 registers and allows some degree of control of its inputs. In particular, the clock and reset inputs of every component within the dashed line can be controlled by the debugging monitor.

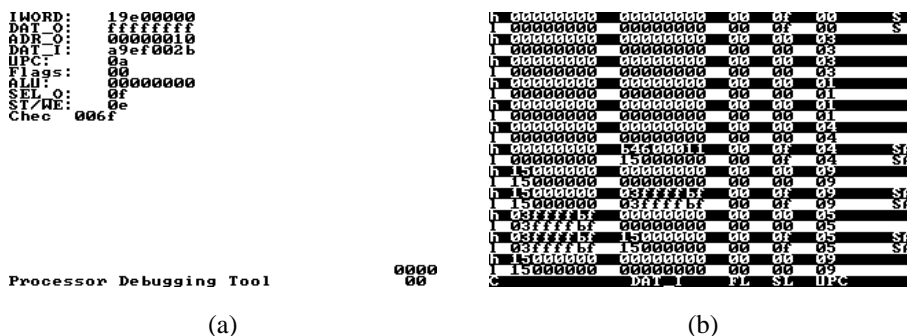


Figure 5.16: Screenshots of the MCGREP-1 debugging monitor, which can be loaded onto an FPGA alongside an MCGREP-1 CPU. (a) shows a dump of the internal MCGREP-1 registers, including IR 0 (*IWORD*), uPC, and Wishbone bus lines (*ADR_O*, *DAT_O*, etc.). (b) shows the ISA interpreter booting up, as viewed through the microprogram debugging mode. Every line represents a half clock cycle and summarises the system state at that time. The first instruction that is fetched is 0x15000000: an *l.nop* instruction (Figure 5.9).

The difficulty with the test methodologies used is the time cost involved: manual intervention is required for generating the RFU microcode, for comparing hardware and simulator operations, and for building each test program. It is clear that a future implementation should aim to reduce this time cost through improved support for automatic testing.

5.3.2 Evaluation against the TARGET parameters

In this section, the capabilities of MCGREP-1 are compared to the parameters of TARGET. The aim of this comparison is to determine how effectively MCGREP-1 can generate instances of TARGET.

MCGREP-1 generates a very restricted subset of all possible implementations of the TARGET architecture. Table 5.5 lists the TARGET parameters: only two of these are configurable in MCGREP-1. It is possible to modify some of the other parameters by extending the MCGREP-

MCGREP-1 Correctness Evaluation

▷ /mcgprep1/testcases

The test corpus used to validate MCGREP-1 CPUs.

▷ /mcgprep1/hw/debug-monitor

VHDL and microcontroller software for the MCGREP-1 debugging monitor.



1 source code: for example, the ISA can be extended by alterations to the method described in section 5.2.2. But there is no way to do this consistently for a range of applications. And other parameters such as the interconnect cannot be configured at all. Therefore, MCGREP-1 is not suitable for experimenting with a range of TARGET implementations. It is not representative of more than one instance of TARGET. This aspect of MCGREP-1 will need to be improved in a future implementation.

5.3.3 Evaluation against the Requirements

As an implementation of TARGET, MCGREP-1 can provide information about how well TARGET meets the requirements of the work (section 3.2) which are:

- Basic block timing invariance (sections 5.3.4 and 5.3.5);
- WCET reduction versus previous work (section 5.3.6);
- Efficient optimisation process (section 5.3.7);
- Scalability, so that WCET reductions can be applied to programs of any size (section 5.3.7).

Support for general software programs has already been demonstrated by the use of benchmark programs in section 5.3.1.

5.3.4 Invariant basic block execution times - calculations

Basic block timing invariance is an essential requirement for implementations of TARGET, because it enables the application of high-quality WCET analysis methods such as IPET (see revised aims, section 3.1).

Two approaches are taken in order to show that CPUs generated by MCGREP-1 have this property. This section describes a method for calculating basic block execution times. The next section (5.3.5) describes an experiment to determine the effects of inter-task interference on MCGREP-1 processors.

Consider Table 5.10. This gives the number of clock cycles required by MCGREP-1 operations. These can be derived from the ISA microprogram generator (section 5.2.2). As the memory latency M is a constant, the exact timing of any basic block can be calculated by summing the cost of every instruction in the basic block. (The cost of every instruction depends on M because of the need to fetch the next instruction during execution.)

Group	Example	Clock cycles
ALU (Immediate)	<code>l.addi</code>	$\max(2, M)$
ALU/Shift (Reg)	<code>l.add</code>	$\max(2, M)$
Compare	<code>l.sfeqi</code>	$\max(2, M)$
Jump	<code>l.bnf</code>	$\max(2, M)$
Load	<code>l.lwz</code>	$4 + 2 \times \max(2, M)$
Move High	<code>l.movhi</code>	$\max(2, M)$
Shift (Immediate)	<code>l.slli</code>	$\max(4, M)$
Special	<code>l.mfspr</code>	$\max(4, M)$
Store	<code>l.sw</code>	$2 + 2 \times \max(2, M)$

Table 5.10: Instruction timings on CPUs generated by MCGREP-1. M is the memory latency in clock cycles.

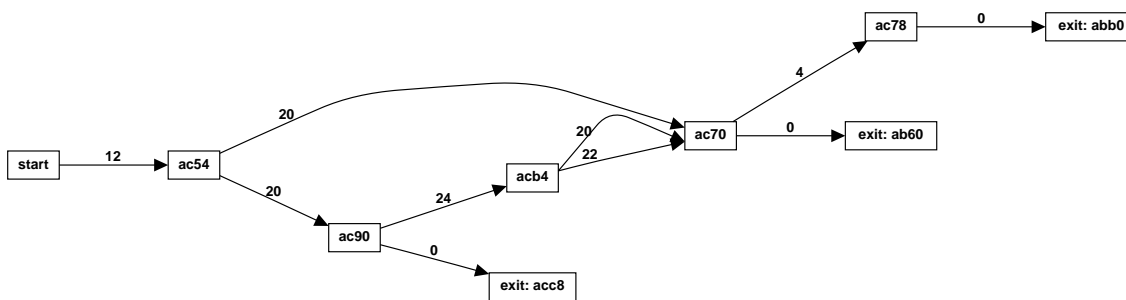


Figure 5.17: Basic blocks in a sample WC path fragment (`III_huffdecode`). Nodes represent conditional branches, and edges represent basic blocks. Each edge is labelled with its execution time cost computed using Table 5.10.

This can be demonstrated by an example that uses code from the `mad` test program. In `mad`, the WC path runs through `III_huffdecode`. The machine code for this path is listed in Figure B.1. In Figure 5.17, each edge represents a basic block on the WC path. The edge labels give the execution time (clock cycles) when running as machine code. It is assumed for the sake of discussion that $M = 0$.

Given the execution time cost of each basic block, it is possible to calculate the machine code WCET by finding the longest path from the start node to any exit node. This path passes through conditional branches at [ac54, acc8, acb4, ac70, ac78] and requires 82 clock cycles. Since MCGREP-1 CPUs have no hidden state, this timing is independent of whatever preceded the start node.

Suppose that this WC path fragment is now converted to a sequence of RFU configurations, encoded as a microprogram. The sequence is dependent on the actions of a human operator during the process shown in Figure 5.11, but one possible arrangement is shown in Figure 5.18. Each basic block in Figure 5.17 is replaced by a sequence of microprogram instructions.

Figure 5.19 shows the effects of microprogramming on the cost of each basic block. The WCET is now 46 clock cycles (including the 2 cycle exit cost). This reflects a reduction in WCET of nearly 1.8 times, without loss of ease of analysis. This is close to the maximum reduction that could be

```

00:  $r4 \leftarrow r6 \gg 0x18$ 
01:  $r12 \leftarrow r4 \& 0xf$ 
02:  $\text{flag} \leftarrow r12 \neq 0$ 
03:  $r29 \leftarrow 1$ , if flag { branch to state 9 }
04:  $r10 \leftarrow 0$ , begin_store( $r16, r10$ )
05:  $r21 \leftarrow r6 \gg 0x14$ ,  $r12 \leftarrow r21 \& 0xf$ , end_store()
06:  $\text{flag} \leftarrow r12 \neq 0$ ,  $r5 \leftarrow 1$ 
07: if flag {  $\text{PC} \leftarrow \text{PC} + 0xab60 - 0xac48$ , return to program }
08:  $r23 \leftarrow 0$ ,  $\text{PC} \leftarrow \text{PC} + 0xab60 - 0xac48$ , return to program
09:  $rT \leftarrow r2 - 0x104$ ,  $r7 \leftarrow r12 \ll 2$ 
10: begin_load( $rT$ )
11:  $r3 \leftarrow \text{end\_load}()$ ,  $r4 \leftarrow r29 \ll r12$ 
12:  $r27 \leftarrow r3 \& r4$ 
13:  $\text{flag} \leftarrow r27 = 0$ 
14: if flag {  $\text{PC} \leftarrow \text{PC} + 0xacc8 - 0xac48$ , return to program }
15:  $r31 \leftarrow r7 + r2$ ,  $r19 \leftarrow r14 - 1$ 
16:  $rT \leftarrow r31 - 0xdc$ ,  $r17 \leftarrow 1$ 
17: begin_load( $rT$ ),  $r15 \leftarrow r17 \ll r19$ 
18:  $r11 \leftarrow \text{end\_load}()$ ,  $r12 \leftarrow r18 \& r15$ 
19:  $\text{flag} \leftarrow r12 = 0$ ,  $r14 \leftarrow r19$ 
20: if flag { branch to state 22 }
21:  $r11 \leftarrow 0 - r11$ ,
22: begin_store( $r16, r11$ ), branch to state 5

```

Figure 5.18: Microprogram sequence for the WC path fragment shown in Figure 5.17, expressed as register transfers. The left hand column shows the clock cycle number.

expected (2 times, as there are two functional units).

5.3.5 Invariant basic block execution times - experiment

By definition, invariant basic block timing ensures that inter-task interference cannot have any effect on task WCET. (This topic was previously discussed in section 2.3.12.) In this section, an experiment is performed to ensure that the actual execution time of a task is always less than or equal to its WCET, regardless of the level of interference from other tasks.

When a high-priority task H preempts a lower priority task T , it could disrupt the CPU state. When execution returns to T , the execution time of each basic block may be changed by the disruption if the CPU lacks the basic block timing invariance property. Figure 5.20 gives a minimal example of conditions for inter-task interference. H and T share a CPU. Under normal circumstances, T is executing. With a period of p , an external interrupt causes H to begin executing. The operating system manages the context switches between T and H .

Let α be the CPU time required to execute T . If basic block timing is truly independent of interference, then α will be the same regardless of the value of p . This is expected to be the case for MCGREP-1 processors, but not for CPUs with hidden state.

1. **Experiment Goal:** To measure the effects of two variables on α , the CPU time taken to execute task T , in the presence of interfering task H .

Two variables are used. The first is the processor platform. Four different processor platforms are tested:

- (a) An OpenRISC soft core CPU, with caches disabled;

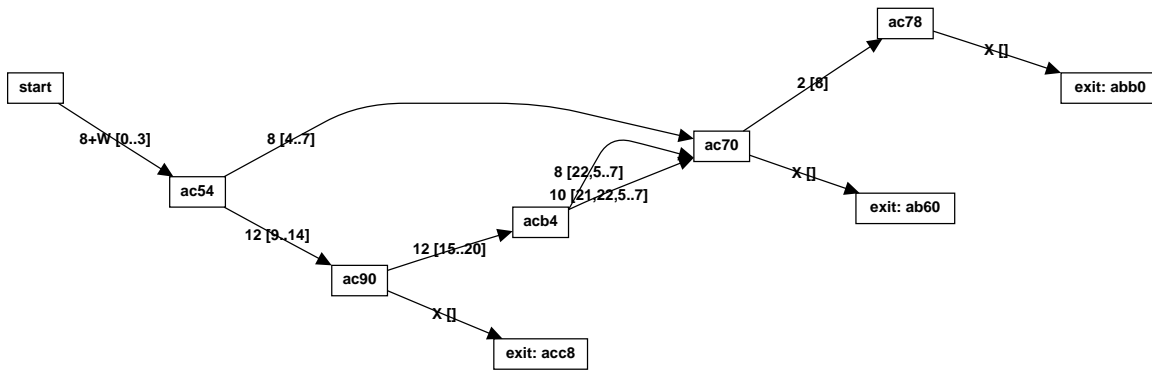


Figure 5.19: Basic blocks as shown in Figure 5.17, but implemented using a microprogram. The graph edges are labelled with the new cost of each basic block, followed by the sequence of microprogram addresses that implement it (see Figure 5.18). X represents the exit cost of returning to machine code: 2 in MCGREP-1 CPUs. W represents the startup cost of launching the custom microcode: 0 in MCGREP-1 CPUs.

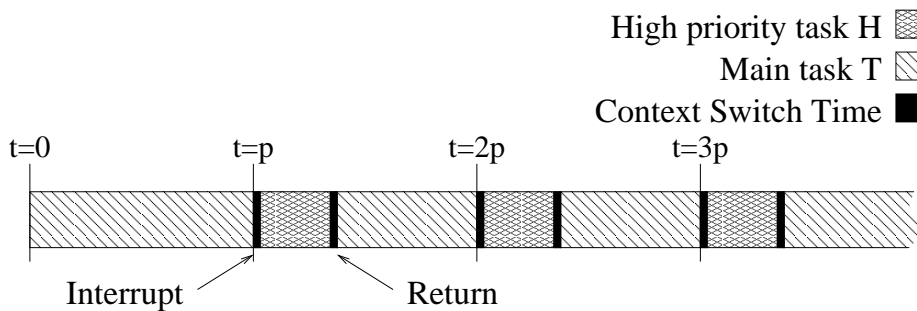


Figure 5.20: Minimal inter-task interference scenario involving two tasks, H and T . H is a periodic task which begins executing at $t = np$ where n is a positive integer.

- (b) An OpenRISC soft core CPU, with 4kb instruction cache enabled;
- (c) An MCGREP-1 soft core CPU, with RFU in use;
- (d) An MCGREP-1 soft core CPU, without RFU extensions.

The second variable is p , the period of high priority task H .

2. **Hardware Setup:** The experimental hardware is a test harness for both OpenRISC and MCGREP-1 CPUs. It includes the following devices:

- A clock cycle counter: this counts the CPU time used by T . The counter can be read and reset by software. The counter is only active when the interrupt enable bit of the CPU machine status register is high [151] - this means that the counter is deactivated while H executes. The timer is 32 bits wide, and is incremented by 1 on the positive-going edge of each clock cycle.
- A countdown timer: this generates an interrupt every p clock cycles, triggering H . The software can program the timer with any value of p .
- A memory latency simulator: this adds additional latency to memory accesses, increasing the cost of a cache miss. In this experiment, the memory latency simulator always adds 12 clock cycles of latency.
- SRAM, interface to host PC, and boot ROM.

The harness can be connected to either an OpenRISC or MCGREP-1 CPU. Therefore, the hardware environment is not a variable.

Unfortunately, the clock cycle counter cannot account for all of the context switching time. Even though the counter is deactivated by the CPU interrupt mechanism, it will always count a fixed number of clock cycles per interrupt. Let this number of clock cycles be β . The effect of this is that $n\beta$ additional clock cycles are added to α for n interruptions. This can be eliminated from the results. β is an (initially unknown) constant for each platform: its value is set by the internal operation of each CPU.

3. **Software Setup:** The experimental software is based on the `aes` benchmark from Table 5.9. The benchmark kernel is wrapped by a procedure that carries out the following functions for each value of p :

- Initialise benchmark software,
- Register *interrupt service routine* (ISR) for H ,
- Program countdown timer with p ,
- Reset clock cycle counter,
- Enable interrupts,
- Run benchmark kernel (task T),
- Store two results:
 - the total measured CPU time for T , $y = \alpha + n\beta$,
 - the total interference count n : the number of times that H executed,
- Disable interrupts.

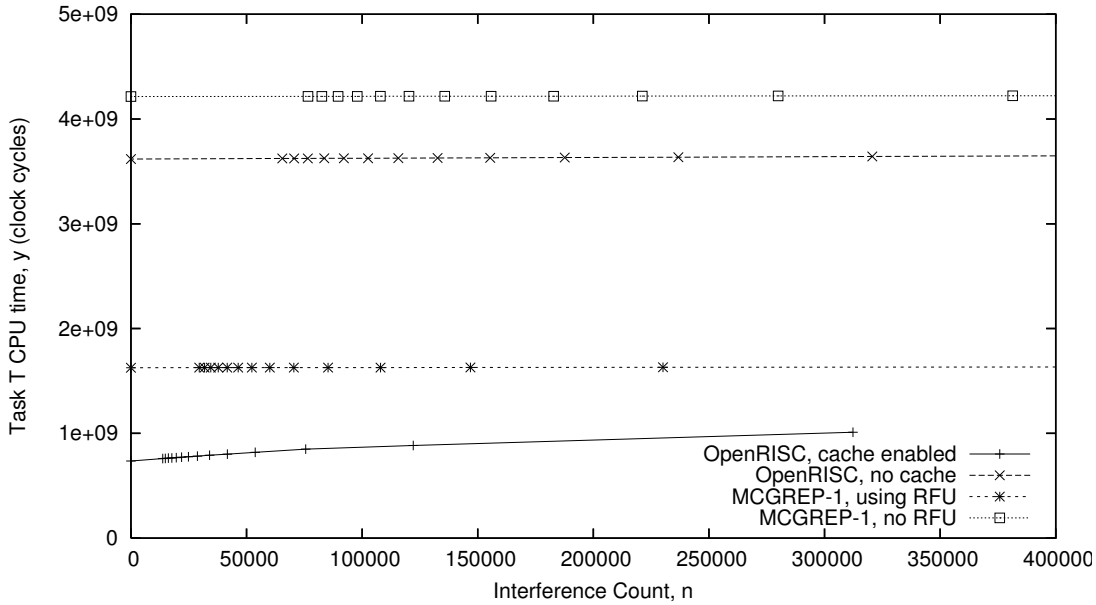


Figure 5.21: The results of the interference experiment, showing the relationship between y and n for each of the four platforms. The relationship is linear for the three uncached platforms. For those platforms, the actual execution time for task T is equal to its WCET regardless of the level of interference from task H . (Raw data appears in Table B.1.)

The task H represents the worst case of an interfering task by invalidating the instruction cache if one is present. The same software is used for CPUs generated by MCGREP-1 and OpenRISC because the two use the same ISA. The only difference is an additional micro-program uploading step for the MCGREP-1 CPU, which is performed before the experiment procedure starts.

4. **Method:** The hardware and software arrangements described above were used to generate results for all four platforms, with values of p ranging from 4000 to 56000 in steps of 4000. An experiment was also performed for each platform with $p = \infty$, i.e. no interference.
5. **Results:** Figure 5.21 shows the results of this experiment, which are also printed in Table B.1. The figure shows the execution times of tasks with varying levels of interference on the four platforms. It is immediately clear that the effect of interference on the cached CPU is much greater than the effect of interference on the other three platforms.

However, the level of interference does appear to have had some effect on the other platforms. It is important to show that this is due to the effects of β , the number of clock cycles required for each context switch. This will be shown by calculating values for α and β for each platform to fit the equation:

$$y = \alpha + n\beta \quad (5.1)$$


α will be constant if basic block timing is independent of interference. The method of least squares [77] is used to calculate values for the two constants for each data set (Table 5.11). The method of least squares minimises the error Π , which is calculated for each platform by

	MCGREP-1 CPU mc only	MCGREP-1 CPU mc+ μ c	OpenRISC cache	OpenRISC no cache
α	4.214e+09	1.626e+09	7.570e+08	3.521e+09
β	3.400e+01	3.400e+01	9.176e+02	1.190e+02
Π	2.956e-12	8.527e-13	2.211e+15	4.547e-13

Table 5.11: Least-squares derivation of α and β for each platform. The fitting error Π is minimised by the method of least squares [77], and is given here to indicate how well each α, β pair fits the results in Table B.1. The fitting error is negligible in both MCGREP-1 cases.

MCGREP-1 Interference Experiment

- ▷ /mcgprep1/testcases/aes/src
Test task T , used for measurements.
- ▷ /mcgprep1/testcases/lib/cache_testing.c
Test harness: carries out measurements, implements task H .
- ▷ /mcgprep1/hw/debug-monitor
VHDL code used for MCGREP-1 measurements.
- ▷ /mcgprep1/hw/openrisc-reference
VHDL code used for OpenRISC measurements.



the equation:

$$\Pi = \sum [y_i - (\alpha + n_i\beta)]^2 \quad (5.2)$$

6. **Evaluation:** The results in Table 5.11 show that Equation 5.1 fits the data very well for both MCGREP-1 cases. There is a linear relationship between the measured time y and the number of interruptions n , due to the overhead of each interruption β . This is demonstrated by the magnitude of the fitting error Π , which is negligible in both MCGREP-1 cases. Therefore, the actual CPU time for task T is invariant on CPUs generated by MCGREP-1.

The method used demonstrates that this property does not hold for OpenRISC with a cache. For that case, the relationship between y and n is non-linear, indicating that α is not constant. This non-linearity is clearly visible in Figure 5.21, and in the magnitude of the least squares fitting error (Table 5.11). The data also shows that the uncached OpenRISC CPU has the same linear behaviour seen with MCGREP-1 CPUs, as expected.

7. **Conclusion:** The results demonstrate that CPUs generated by MCGREP-1 have invariant basic block execution times in both RFU mode and ORBIS32 ISA mode. This property is shared with the OpenRISC processor operating in cacheless mode: the measured execution time of task T is equal to its WCET, regardless of interference from task H . In contrast, when a cache is in use, interference from H does cause the measured execution time to increase.

5.3.6 WCET Reduction versus Previous Work

MCGREP-1's method of optimising WC path fragments is unusual because of its predictable operation. Approaches for optimising general software execution often rely on caches (e.g. in a VLIW CPU [87]) or dynamic superscalar issue (e.g. in a x86 CPU [135]), but these do not meet the requirements (section 3.2), so they are not directly comparable.

MCGREP-1 CPUs are directly comparable with CPUs that use scratchpads or locked caches in place of caches, because these can also preserve the basic block timing invariance property. Two such CPUs are OpenRISC and Microblaze, both soft cores that may be used within embedded hard real-time systems. For the purposes of the following experiments, it is assumed that the CPU is connected to very large instruction and data scratchpads, so the effective RAM latency M is 1 clock cycle. In this arrangement, all of the CPUs have basic block timing invariance.

WCET reductions can be tested using benchmark programs such as those listed in Table 5.9. Ideally, the WCET reduction would be computed by analysis. However, the benchmarks do not support this because execution constraints are not available. Therefore, an assumption is made to allow them to be used: each test program is assumed to be a single-path program. Indeed, this is the case when the input data is fixed, and for the benchmark programs listed in Table 5.9, an input data set is specified for each program by the benchmark designer [108, 154]. As Puschner has observed [204], the WCET of a single-path program can be obtained by measurement. This assumption provides a quick way to use the existing benchmark programs to determine possible WCET reductions.

The benchmarks from Table 5.9 will not provide complete information about the WCET reductions available using MCGREP-1. No set of benchmarks can be representative of all types of program for an embedded system. However, the benchmarks do provide examples of possible WCET reductions, and since the source code and input data is standardised, comparison is also possible with the Microblaze and OpenRISC CPUs.

1. **Experiment Goal:** To compare the WCET of programs on CPUs generated by MCGREP-1 with other simple CPUs, assuming that large scratchpads are used in place of external RAM.

There are two variables. The first is the benchmark program: this is taken from Table 5.9. The second is the processor platform. Four platforms are tested:

- (a) An OpenRISC soft core CPU;
- (b) A Microblaze soft core CPU;
- (c) An MCGREP-1 soft core CPU, with RFU in use;
- (d) An MCGREP-1 soft core CPU, without RFU extensions.

2. **Hardware Setup:** For this experiment, the following hardware components are needed:

- A clock cycle counter: this counts the CPU time used by any program. A 32 bit counter is used. The counter is incremented once per clock cycle.
- An instruction counter: this is incremented every time the CPU fetches an instruction. This allows *instructions per clock cycle* (IPC) values to be computed.
- Scratchpad RAM, an interface to a host PC, and a boot ROM. (The scratchpad RAM is implemented using external RAM - the clock frequency of the core is scaled appropriately to mimic the low latency of a real scratchpad.)

Benchmark	OpenRISC/MCGREP-1 CPU instructions	Microblaze instructions
aes	1.91e+06	1.86e+06
crc32	3.94e+06	3.94e+06
dijkstra	2.81e+08	1.73e+08
g721	3.37e+08	2.78e+08
jpeg	7.36e+06	5.53e+06
mad	5.58e+07	3.34e+07
qsort	5.01e+06	5.18e+06
sha	7.01e+07	5.50e+07

Table 5.12: Number of instructions executed for each benchmark.

These components are connected to a harness for Microblaze, OpenRISC or an MCGREP-1 CPU. All of these CPUs are configured without caches, and with hardware multiplier and barrel shifter support. The RAM latency $M = 1$.

3. **Software Setup:** For this experiment, a simple benchmark procedure is used to time the execution of each benchmark program. The steps that are carried out are as follows:
 - Initialise benchmark software,
 - Reset counters,
 - Run benchmark kernel,
 - Output counter values.

During this procedure, interrupts are always disabled.

4. **Method:** The benchmark procedure described above was applied to each benchmark on each platform.
5. **Results:** Figure 5.22 shows the execution times of each benchmark on each platform, using raw data printed in Table B.2. The values are normalised to the execution time of the benchmark on an MCGREP-1 CPU in “machine code only” mode. Presenting the results in this form accounts for the ISA differences between Microblaze and OpenRISC: some benchmarks require more instructions on one CPU than the other. Figure 5.22 avoids this problem by considering the overall function of the benchmark only.

Table 5.12 shows the number of instructions executed for each benchmark on each platform. This can be used to compute the IPC value during each benchmark execution. Mean average IPC values are shown in Table 5.13.

6. **Evaluation:** Results in Figure 5.22 and Table B.2 indicate that programs running on the MCGREP-1 CPU can have lower execution times than either OpenRISC or Microblaze when using custom microcode. This is true for all but one member of the benchmark set (`jpeg`), where Microblaze achieves a lower execution time for the benchmark.

One explanation for this problem is ISA differences. Table 5.12 gives the instruction count for `jpeg` on Microblaze and OpenRISC. The Microblaze count is 33% smaller: therefore, an

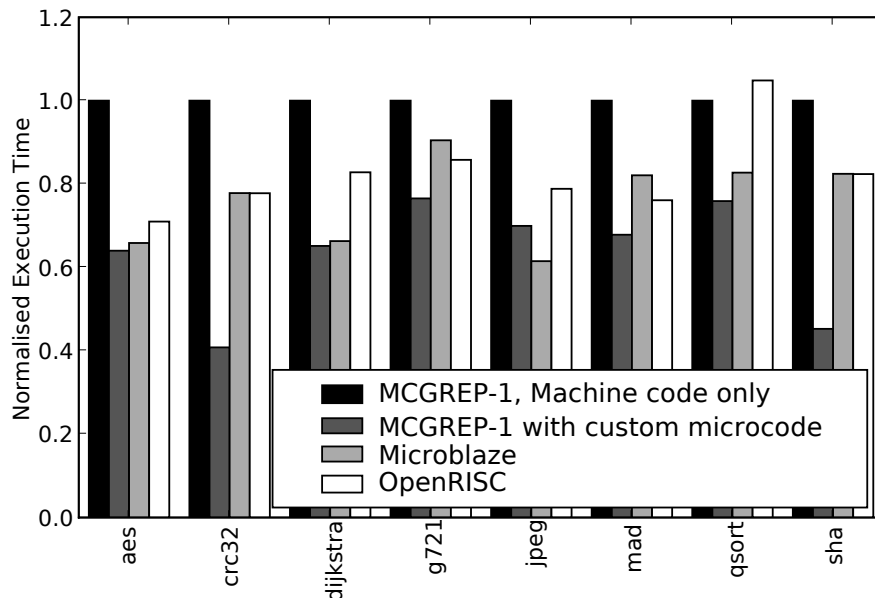


Figure 5.22: Normalised benchmark execution times on four platforms. As these are considered to be single-path programs, the measurements are equivalent to WCETs. (Raw data appears in Table B.2.)

	MCGREP-1 mc only	OpenRISC no cache	Microblaze no cache	MCGREP-1 mc+μc
Mean IPC	0.261	0.319	0.273	0.431

Table 5.13: Mean IPC for each CPU as measured during benchmark execution with $M = 1$. The IPC for CPUs generated by MCGREP-1 is calculated using the notional number of instructions that would be executed in the absence of custom microcode: i.e. the OpenRISC figure from Table 5.12.

MCGREP-1 Execution Time Experiment

▷ /mcgprep1/testcases

Test cases used for measurements.

▷ /mcgprep1/hw/debug-monitor

VHDL code used for MCGREP-1 measurements.

▷ /mcgprep1/hw/openrisc-reference

VHDL code used for OpenRISC measurements.



MCGREP-1 CPU is put at a disadvantage for this benchmark by the ORBIS32 ISA, or the compiler that generated the jpeg machine code.

In general, an MCGREP-1 CPU has poorer performance than both OpenRISC and Microblaze when working in “machine code only” mode. (The `qsort` benchmark is an exception.) The IPC figures (Table 5.13) indicate that an MCGREP-1 CPU is (on average) 22% slower than OpenRISC during the benchmarks. However, when custom microcode is used, an MCGREP-1 CPU operates faster than OpenRISC.

7. **Conclusion:** MCGREP-1 CPUs are not as fast as Microblaze or OpenRISC when executing machine code with $M = 1$. However, such CPUs can complete benchmarks in less time than OpenRISC when custom microcode is used. Figure 5.22 shows the magnitude of improvement that is possible. This demonstrates that MCGREP-1 CPUs can reduce program execution times, and for the specific case of the benchmarks examined here, this result also applies to WCETs.

The benchmark experiments uncovered one case in which the Microblaze CPU was faster than an MCGREP-1 CPU. This is likely to be due to ISA differences. Table 5.12 suggests that the Microblaze ISA is more efficient than the ORBIS32 ISA because some benchmarks requires fewer instructions to complete on Microblaze. This suggests that the choice of the ORBIS32 ISA may require a review.

5.3.7 Other Requirements

CPUs generated by MCGREP-1 do meet some of the scalability requirements:

- The CPUs are able to scale to support any number of tasks that require access to the RFU, because the microprogram store can be reconfigured. Additionally, each task can be partitioned into regions using schemes described by previous work (section 4.3.4). Currently, the MCGREP-1 tools allow programs to be partitioned manually, with calls to the Upload procedure (section 5.2.4) being made at each region boundary. In an automatic scheme, such “copy points” would be placed by a tool aiming to minimise the WCET [201]. This approach is scalable to any number of tasks, and any size of task, provided that large and complex tasks can be partitioned.
- The CPUs make efficient use of FPGA space. Table 5.14 compares OpenRISC, Microblaze and MCGREP-1 cores on a Spartan-3 FPGA. The sample MCGREP-1 CPU is about 40%

Core	LUTs	FFs	BRAMs	Mults.	Max. Freq (MHz)
MCGREP-1	2756	228	14	3	36.8
Microblaze	1315	480	0	3	78.1
OpenRISC	3817	1095	2	3	53.0

Table 5.14: Sizes and maximum speeds of some 32-bit soft cores, synthesised with identical settings for a large Spartan-3 FPGA (part id `xc3s5000-fg1156-4`). All cores were configured with near-identical features where possible (no cache, no FPU, no MMU, and hardware multiplier). Three hardware multipliers (Mults.) are needed by each CPU to create a 32-bit multiplier.

smaller than OpenRISC in terms of combinational logic (LUTs). It also uses fewer flip-flops (FFs). However, it uses many more block RAMs (the actual number is dependent on the microprogram store size parameter), and Table 5.14 also shows that the maximum clock frequency of the MCGREP-1 CPU is about 40% lower than that of OpenRISC. The Microblaze CPU is faster and smaller than both OpenRISC and MCGREP-1 CPUs: this is due to optimisations for implementation on Xilinx FPGAs.

But MCGREP-1 does not meet the requirement for an efficient optimisation process because human intervention is required. A developer must identify and hand-optimize WC path fragments to reduce the WCET. A new implementation must address this problem, because it is a severe restriction on the range of experiments that can be carried out. Additionally, other limits apply to MCGREP-1's scalability due to the nature of the design. High-level limits include:

- The degree of ILP available - at most 2 with two functional units;
- The size and complexity of each WC path fragment - this is limited by the size of the microprogram store;
- The ORBIS32 instruction set appears to be less efficient than the Microblaze instruction set (Table 5.12), but the MCGREP-1 design does not allow the ISA to be easily changed, so it is difficult to take advantage of this.

There are also some low-level limitations of the MCGREP-1 microarchitecture and related tools:

- There is no register between the instruction decoder and the control unit (Figure 5.7). This lowers the maximum clock frequency (Table 5.14) as uPC computation is dependent on instruction decoding, and both must be done within a clock cycle.

Assuming low memory latency, the higher speed of Microblaze eclipses the performance gain from an MCGREP-1 CPU: one could use an 80MHz Microblaze in place of a 40MHz MCGREP-1 CPU. But a redesign is required to increase the clock frequency of CPUs generated by MCGREP-1, as it is not possible to add extra pipeline registers without forcing a rewrite of the microprogram and control unit.

- Every ORBIS32 instruction and every microinstruction requires two clock cycles because of the register port limitation described in section 5.2.3.7. This does not make optimal use of the resources available - in particular, each ALU is idle for 50% of the time.

- The MCGREP-1 simulator is very slow, executing around 10k instructions per second on present (2008) workstations. This is fast enough for basic functional tests and short benchmarks, but makes more extensive testing intractable.

All of these points need to be addressed by a better design.

5.3.8 Summary

MCGREP-1 has been shown to work correctly (section 5.3.1). CPUs generated by MCGREP-1 meet the following requirements (section 3.2):

- Full support for general software: demonstrated in section 5.3.1.
- Basic block timing invariance: demonstrated in sections 5.3.4 and 5.3.5.
- WCET reduction versus previous work: demonstrated in section 5.3.6.

But the requirement for an efficient optimisation process has not been met, and the scalability of the MCGREP-1 design is limited. All but one of the scalability issues described in section 5.3.7 are software or hardware engineering problems that can be solved by improvements to existing MCGREP-1 hardware. For example, the pipeline issue regarding the instruction decoder and control unit can be resolved by addition of a pipeline register.

The MCGREP-1 requirement for hand optimisation is different, because it acts as a barrier preventing automatic searches for the best way to reduce the WCET of a program. An automatic optimiser is required for the next version, along with solutions to the engineering issues listed in section 5.3.7. Additionally, as section 5.3.2 describes, the subset of TARGET generated by MCGREP-1 is very small. The new design must allow a greater proportion of the design space to be explored.

The next chapter describes the implementation of a second iteration of the TARGET generator, MCGREP-2, to address these issues.

Chapter 6

Extensible Implementation

MCGREP-1, the CPU generator described by chapter 5, demonstrated most of the properties required for TARGET. Reduction in WCET was shown (section 5.3.6) along with basic block timing invariance (sections 5.3.4 and 5.3.5). A mechanism to support unlimited numbers of WC path optimisations was discussed (section 5.3.7).

However, the MCGREP-1 generator lacked scalability in many ways (section 5.3.7). The scalability problems force a redesign to better meet the requirements for the work. The goal of this chapter is to describe the second implementation of a generator for TARGET, named MCGREP-2. As in the previous chapter, the aim is to use the generator to evaluate a subset of TARGET, but because MCGREP-2 has better scalability and flexibility, a larger subset can be considered.

The subset will be evaluated as part of the abstract WCET reduction process shown in Figure 6.1, where it will provide the features of the three modules highlighted in bold. The following new information will be incorporated into the design process, based on problems uncovered during the implementation of MCGREP-1 (section 5.3.8):

- Automatic microcode generation for the RFU is essential, to allow experiments to be carried out with different TARGET parameters and facilitate automatic WCET reduction. Section 6.1 gives some background on applicable techniques for microcode generation, with an implementation described in section 6.3.
- Improved parameterisation support must be built into the CPU generator tools from the start, which should support changing as many of the TARGET parameters (Table 4.1) as possible. In particular, the ability to change the array size (i.e. l , m and n) is important for experiments. The limitations of MCGREP-1 force a new implementation of the components in the storage and execution levels of Figure 6.1. This is described in section 6.2.1.
- The performance of the simulator component must be improved. This is discussed in section 6.2.5.

The complete MCGREP-2 system is evaluated in section 6.4.

6.1 Automatic Microcode Generation Methods for TARGET

Many methods could be used to map code to the TARGET RFU (Figure 4.5). The important factors to consider are the input, the method used, and the type of RFU assumed by the output. The

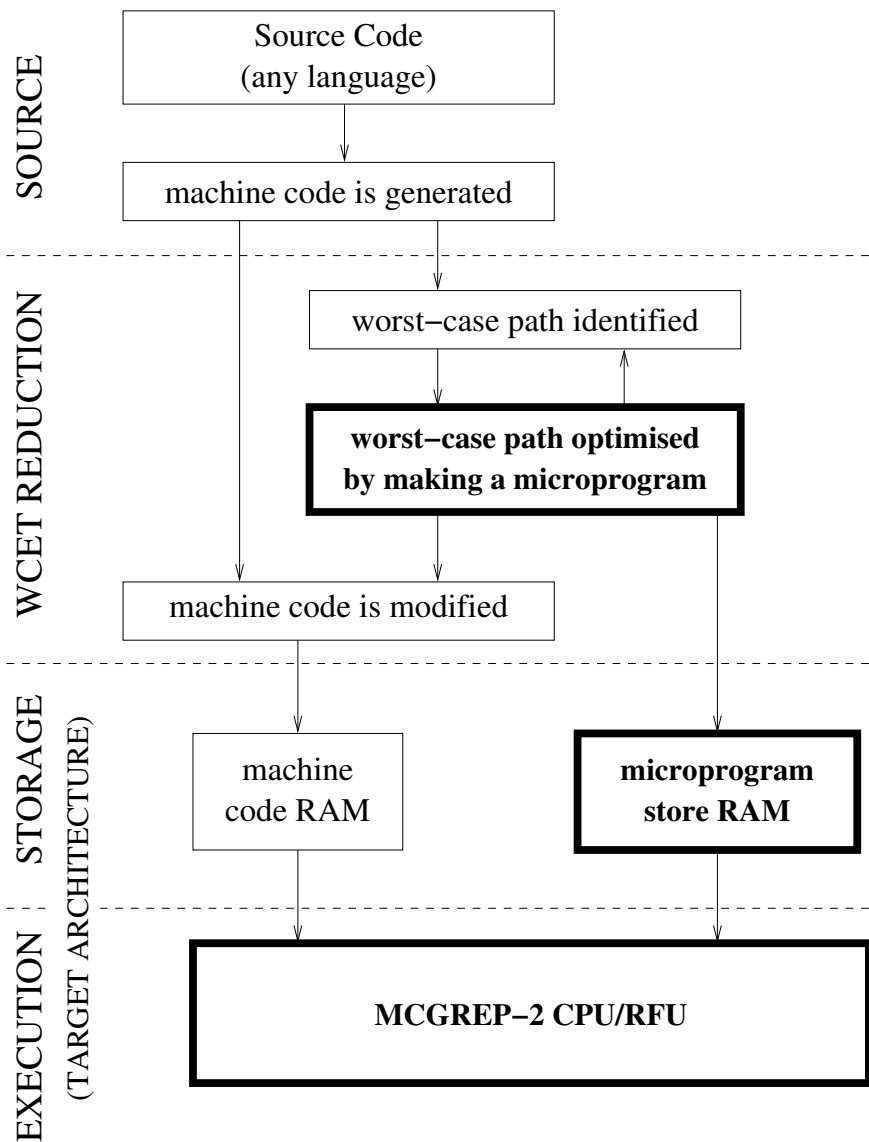


Figure 6.1: The components of the abstract WCET reduction process that are implemented in this chapter are highlighted in bold.

requirements for TARGET specify machine code as the input because full support for general software programs is required, independent of the software language or compiler (section 3.2).

Because the RFU and the microcode generator are being defined together, the type of RFU can be adjusted to meet the requirements of the generator. As discussed in section 4.2.2, the abstract TARGET RFU is a connected array of w functional units. Within the CGRA research field, a variety of automatic methods have been used to generate microcode, as described in section 2.6.6. However, optimal use of a CGRA requires a custom data flow language because software languages (including machine code) cannot make adequate use of the parallelism available within the CGRA. Therefore, CGRA code generation algorithms are not suitable when the input is machine code.

Automatic microcode generation and RFU configuration generation are closely related problems (section 2.6.8): a carefully designed RFU can facilitate automatic microcode generation. Introducing the topic of user microprogramming revealed previous work on the issue of microprogram compaction, which is applicable to architectures which allow ILP to be exploited. As described in section 2.6.8, two general forms of microprogram compaction are possible:

- Local compaction [152] is limited by basic block boundaries, which restricts the ILP that can be exploited because speculation is not possible [268]. Local compaction will not satisfy the requirement to reduce WCET beyond what is possible using previous work (section 3.2) because existing work already makes use of ILP within a basic block to reduce WCET while maintaining predictable timing [217].
- Global compaction is more powerful, because it enables basic block boundaries to be disregarded. A search process can be used to carry out global microprogram compaction [253], but this hits the same types of problems faced by place and route algorithms for FPGAs because locally suboptimal compaction decisions may need to be made in order to reach a global optimum. Fortunately, it is possible to carry out global compaction without this type of search [85].

For a variety of reasons, including the limitations of user microprogramming for general-purpose computers, and the success of the RISC paradigm (where static CPU optimisations in the form of complex microprogrammed instructions are replaced with dynamic CPU optimisations provided by a cache [144]), continued development of global microprogram compaction algorithms has been focused on VLIW CPUs. In principle, the function of each compaction algorithm is the same, but the target is machine code for a VLIW CPU rather than a microprogram. Using VLIW terminology, global compaction algorithms are commonly known as *instruction scheduling* algorithms, although some authors regard “instruction compaction” as more closely representing the nature of the process [87].

In the following sections, instruction scheduling algorithms are reviewed (sections 6.1.1 to 6.1.3) and then compared to the requirements for the TARGET RFU in section 6.1.4.

6.1.1 Generating VLIW Code

The VLIW approach (section 2.2.2.3) is embodied by static exploitation of ILP within a program. The ISA of a VLIW CPU allows the compiler to specify two or more operations to be carried out simultaneously for each instruction. In a VLIW CPU, each instruction (or *bundle* [85]) is a container for several operations.

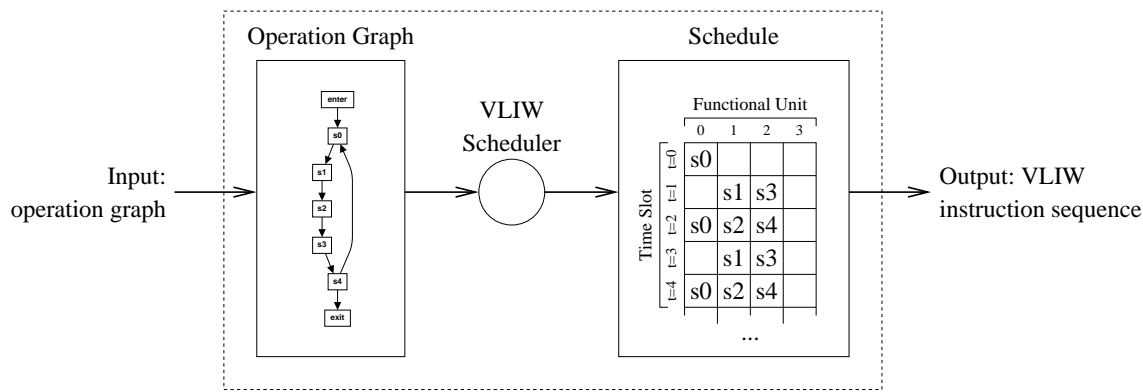


Figure 6.2: Overview of the operation of all VLIW schedulers. The input may be an execution path graph (section 2.1.1) or a data flow graph, with nodes or edges representing sequences of microoperations. These could represent an entire program, a hotspot, or a WC path fragment, although schedulers often place restrictions on the characteristics of this graph. The output is a schedule, a two-dimensional allocation of operations to hardware resources, with the dimensions being “functional units” and “time slots”. There is one time slot per VLIW instruction, so the schedule maps readily onto a sequence of VLIW instructions.

A VLIW compiler is similar in function to a conventional CPU compiler (e.g. for x86). The differences are introduced in the later stages of compilation: the ISA-level code generation and optimisation. These stages are generally more complex within a VLIW compiler, because low-level parallelisation is not easy to specify. Handling control flow efficiently without losing ILP is the main difficulty, which motivates global scheduling algorithms that operate across basic block boundaries.

Finding general optimal solutions to scheduling problems is NP-hard [104], so algorithms usually employ heuristics to obtain solutions in a reasonable time. One class of heuristics for global scheduling are those used by issue units in superscalar out-of-order CPUs [236], which dynamically map an incoming instruction stream to functional units. These heuristics must operate in $O(n)$ time for a stream of n instructions, and can often obtain a close approximation to an optimal schedule. Although VLIW schedulers do not all operate in $O(n)$ time, efficiency is normally very high. (This is evidenced by compile time statistics as seen in [54]).

In [87], Fisher divides VLIW schedulers into two classes (acyclic and cyclic) according to the type of code that they generate. This classification is reused here. Acyclic (“trace”-style) schedulers are described in section 6.1.2, and cyclic (modulo) schedulers are described in section 6.1.3. The procedures applied by these schedulers are considered to have the general form shown in Figure 6.2.

6.1.2 Acyclic Scheduling

An *acyclic scheduler* conventionally attempts to minimise the ACET for a program, by assuming that branches within the program are always taken in the most probable direction. The input is usually a loop-free subgraph of the entire program, represented as an execution path graph. A profile is often used (section 2.4.1) to indicate which paths through the subgraph are most probable: the use of this information is feedback-directed optimisation (section 2.4.2). This class of scheduler could

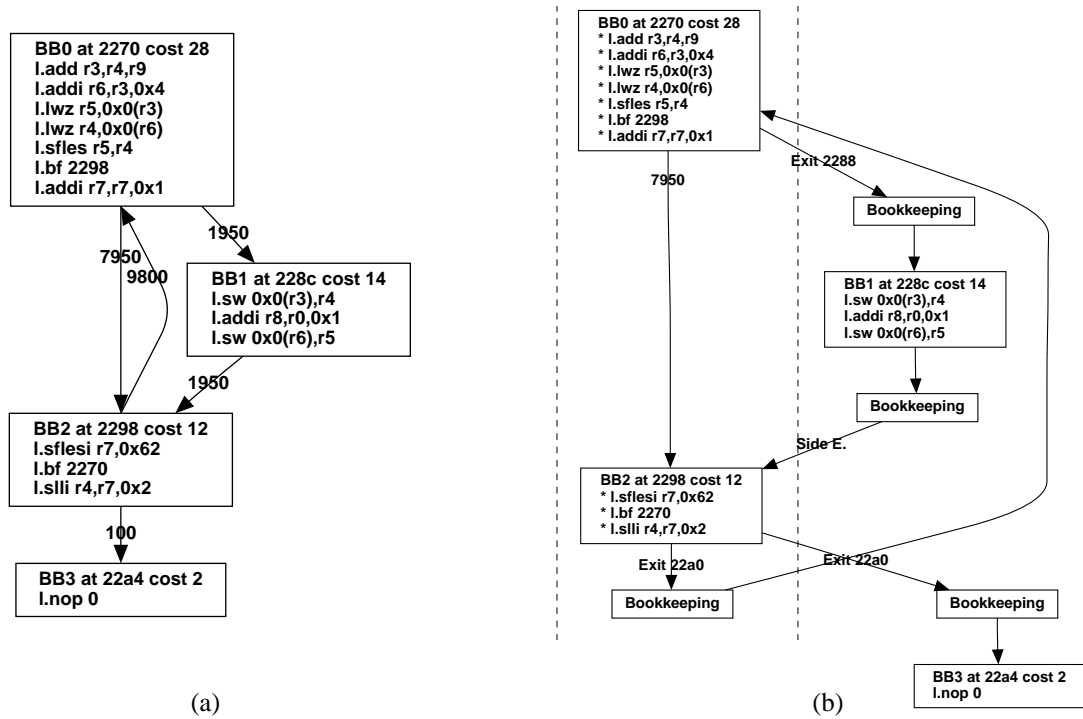


Figure 6.3: Trace scheduling example. (a) is the trace scheduler input: a basic block graph of the bubble sort inner loop (Figures 7.7 and B.2). Edges are labelled with an execution count obtained by profiling, and nodes are labelled with ORBIS32 assembly code (Appendix C). (b) shows the path selected by a trace scheduler within the two dashed vertical lines, and also shows the bookkeeping nodes inserted to handle trace exits and side entrances (marked as “Side E.”).

be called “trace”-style because all acyclic algorithms are related to the *trace scheduling* algorithm described by Fisher [85].

Trace scheduling is best illustrated by an example. Figure 6.3(a) is a basic block graph of the inner loop of a bubble sort program. (The C source of this program appears in Figure 7.7, and an ORBIS32 assembly listing is in Figure B.2.) In the basic block graph, each edge represents a transition between two basic blocks. Each edge is labelled with the number of times that the edge was executed during a sample execution of the bubble sort program.

Given this input, trace scheduling proceeds as follows [85, 87]:

1. Trace Formation:

The trace scheduler selects the most likely path through all basic blocks marked as “uncompacted”. (Initially, all basic blocks are uncompacted.) Conventionally, the most likely path is found using edge profile data, as shown in Figure 6.3.

In this case, there are two possible paths through the trace. A comparison in BB0 (basic block 0) tests whether two elements need to be swapped by the sort procedure. If they do, then BB1 is executed. Otherwise, BB2 is executed. Profile data shows that this happens $\frac{1950}{1950+7950} = 20\%$ of the time: the input to the sort algorithm is mostly in the correct order. So the most likely successor to BB0 is BB2. The trace formation procedure selects this path, then inserts “bookkeeping” nodes on every edge that joins or leaves that path, such as the

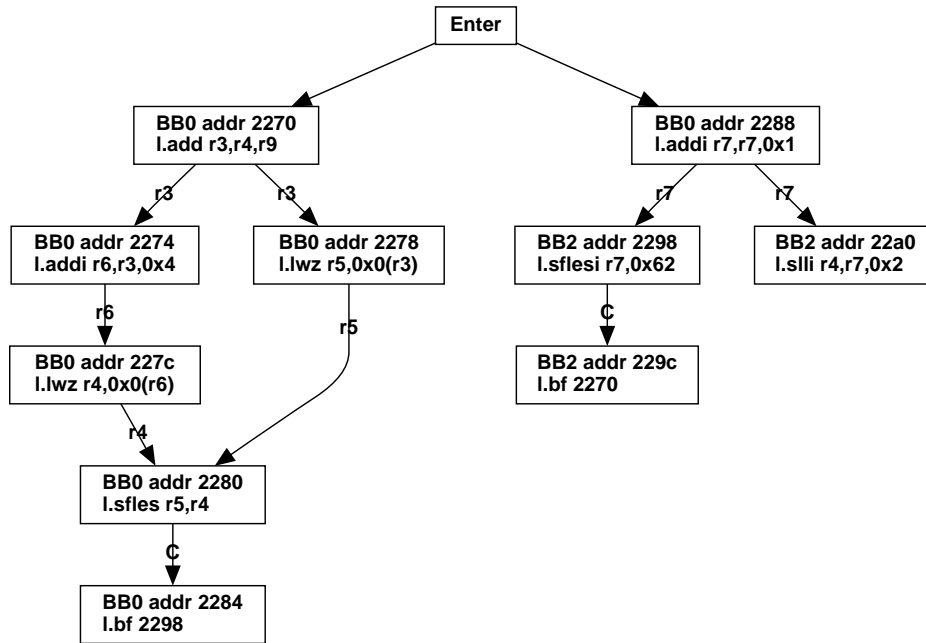


Figure 6.4: Data flow graph for the trace example given in Figure 6.3(b). The graph edges represent the movement of data between instructions, and each one is labelled with the ORBIS32 register used for that data by the original assembly code. New register assignments are made during scheduling to remove false dependences. Dependences that enter and leave the trace are not all shown in order to improve the clarity of the graph.

edges to BB1.

2. Operation list building:

Operations are selected from the trace path in dependence order. This is arranged by building a data flow graph from the operations in the trace (as illustrated in Figure 6.4), then sorting the operations into a list based on their distance from the root of the data flow graph (Figure 6.5). Register renaming [2] may take place during this step, reducing “false” dependences that exist only because a physical register is reused.

3. Compaction:

During this part of the process, the operation list is examined in order. Each operation is selected and compacted into the time slot at the end of the output schedule, as illustrated in Figure 6.5. The operations of the original program (Figure 6.3(a)) are rearranged to exploit ILP, without regard for branches. The algorithm never backtracks.

Each schedule time slot represents an instruction, with one or more VLIW operations embedded within it. It is possible that an operation will not fit in a particular time slot, either as a result of data dependences being unmet, or because the hardware resources required are not available. In this event, it is always possible to fit an operation into the next (empty) time slot, so the trace scheduler advances to the next slot and tries again.

Operation Table

D	Operation	D	Operation
4	l.add r3,r4,r9	11	l.slli r4,r7,0x2
5	l.addi r6,r3,0x4	12	l.bf 2270
6	l.addi r7,r7,0x1	12	l.lwz r4,0x0(r6)
8	l.lwz r5,0x0(r3)	13	l.sfles r5,r4
11	l.sflesi r7,0x62	14	l.bf 2298

Compacted Operations

Cycle	Unit 0	Unit 1	Unit 2
1	l.add r3,r4,r9	l.addi r7,r7,0x1	
2	l.addi r6,r3,0x4	address gen	l.sflesi r7,0x62
3	address gen	l.slli r4,r7,0x2	l.lwz r5,0x0(r3)
4	exit 22a0	l.add r3,r4,r9	l.lwz r5,0x0(r3)
5	l.addi r6,r3,0x4	address gen	l.lwz r4,0x0(r6)
6	address gen	l.addi r7,r7,0x1	l.lwz r4,0x0(r6)
7	l.sfles r5,r4		l.lwz r5,0x0(r3)
8	exit 2288		l.lwz r5,0x0(r3)
9	l.sflesi r7,0x62	l.slli r4,r7,0x2	l.lwz r4,0x0(r6)
10	exit 22a0		l.lwz r4,0x0(r6)
11	l.sfles r5,r4		
12	exit 2288		
13	bookkeeping r3		
14	bookkeeping r4		
15	bookkeeping r5		
16	bookkeeping r6		
17	bookkeeping r7		

Figure 6.5: The trace scheduling compaction process, for the trace example given in Figure 6.3(b). A list of operations (top) is sorted into data flow order, then moved one by one into the output schedule (bottom). Load operations (l.lwz) require two microinstructions.

4. **Bookkeeping and Exit Cleanup:**

The resulting trace schedule replaces the original sequence of basic blocks. Code is then built for the bookkeeping nodes, to ensure that execution flow along graph edges that lead into the trace after the beginning operate correctly. Because a trace can rearrange operations and rename registers to expose ILP, transitions leaving a trace must ensure that the register file is in the state expected by the target code. This is the job of the bookkeeping algorithm. The process is complicated by the need to support *side entrances*, labelled “Side E.” in Figure 6.3(b). These are places where control flow joins a trace after its beginning [87].

5. **Repeat as necessary:**

The trace scheduling process is repeated while uncompact blocks exist. The end result is that all basic blocks in within a program have been compacted with traces, although some of these might only involve a single basic block.

Trace scheduling was found to be far more effective than local scheduling approaches or earlier ad hoc global scheduling attempts [85]. This is because it prioritises the most likely sequence of operations, and schedules this sequence on the available resources without backtracking, ignoring the requirements of side entrances and exits. Provided that the CPU always executes the most likely sequence of operations, the control flow never has any effect on ILP. If a different sequence is used (e.g. if two elements require swapping in the bubble sort program), it is handled by another trace, with a time penalty introduced by the bookkeeping code.

In the example presented, trace scheduling is extremely effective because BB0 is usually followed by BB2. The branches within the trace are said to be *biased* towards a particular path (BB0 → BB2), so the most significant bookkeeping penalty is on the loop back edge, and this can be reduced by unrolling the loop. In every case, the advantages of trace scheduling over earlier methods of acyclic scheduling include:

- Operations can be moved across basic block boundaries, so the maximum ILP within a trace can be found. Local compaction approaches could only rearrange operations within one basic block.
- Trace scheduling is fast, as backtracking is not used and there is no form of iterative improvement process (as applied by earlier global scheduling approaches [253]). Therefore, time complexity is bounded.
- The execution path graph may be treated as a subgraph of a larger graph. Each subgraph can be scheduled separately, and can have multiple entrances and exits. This, combined with the algorithm’s efficiency, permits the trace scheduling algorithm to scale up to schedule any subset of the program.
- The space used by the output schedule is minimised because operations are only duplicated for bookkeeping purposes.

Its weaknesses are as follows:

- Trace-style schedulers do not handle control-intensive code well unless execution probabilities are biased towards a particular path. The costs of moving between paths (bookkeeping) can be high.

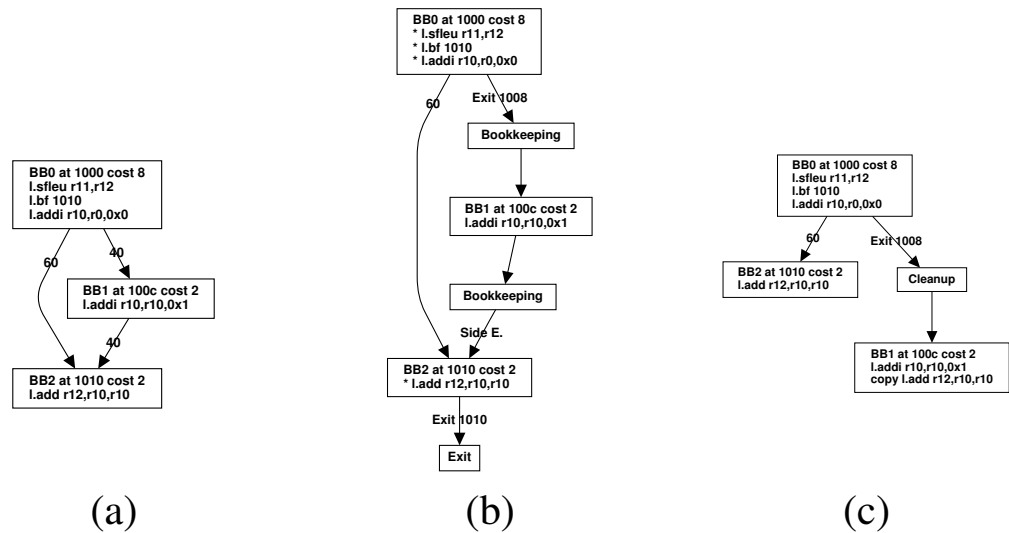


Figure 6.6: An example of tail duplication. Program (a) contains a conditional operation (basic block 1) which executes 40% of the time according to profile data. Therefore, trace scheduling puts it in a separate trace, (b). But this means that bookkeeping operations are often used: twice per execution. When superblock scheduling is used, the tail of the first trace is duplicated, as (c) shows. This avoids any need to reenter the first superblock.

- Loops are not well supported: trace-style schedulers use loop unrolling, which results in code expansion.
- Bookkeeping operations are difficult to generate correctly. Operations may have been executed speculatively in one flow. Register allocations may also be different. Complex bookkeeping algorithms must be used to manage these issues and generate correct cleanup code. According to [85]:

“The details of the bookkeeping phase are very complex and their formal presentation is unintuitive.”

While a correct implementation of bookkeeping has been demonstrated, the additional complexity is undesirable.

- Each node in the execution path graph is only optimised once, for one execution path. This strategy does not efficiently account for basic blocks that are shared between two likely execution paths.

Despite these disadvantages, trace scheduling is a scalable algorithm for exploiting ILP. The weaknesses are addressed by later developments:

- **Superblock scheduling** [48, 47] is a restricted form of trace scheduling in which side entrances are not permitted. Instead, *tail duplication* is used (Figure 6.6). This type of trace is known as a “superblock”.

Superblocks eliminate the need for the most complex form of bookkeeping, as it is only possible to enter each superblock at the beginning. Bookkeeping (“cleanup”) code is still needed at superblock exits, to prepare for the next superblock, but this code is relatively simple because it does not need to map the state of one partially executed trace onto another partially executed trace.

This approach is better able to handle programs with a large amount of control flow. This is firstly because fewer bookkeeping operations are required, and secondly because multiple paths can pass through the same basic blocks without causing a break in the trace. However, more code is generated due to the use of tail duplication.

- **Hyperblock scheduling** [170] is an enhancement of superblock scheduling in which partial *if conversion* [3] is used to collapse some control flow into a linear trace. In programs without heavily biased control flow, this reduces the overhead introduced by moving from one superblock to another, because a direction does not have to be assumed for every branch.

The disadvantage is that unexecuted operations can take up space within each hyperblock, which is a particularly severe problem if one path is longer than another. Heuristics can be used to determine whether control flow should be handled by creating a new hyperblock or by using if conversion. This process is conventionally guided by probability annotations: conversion is not used on unlikely execution paths.

Predication and speculative execution are brought together by hyperblock scheduling, but the two do not fit well together because all predicated operations depend on a predicate source. To offset this problem, hyperblock scheduling permits predicated operations to execute speculatively before their predicates are computed. This is termed “instruction promotion” [170], as conditional operations are effectively promoted to the scope outside their `if` statement. An inverse operation, “instruction merging”, is used whenever identical instructions execute with complementary predicates.

Hyperblock scheduling also includes “node splitting” operations which are applied to reverse the effects of if conversion when control flows merge. Nodes following the merge point are duplicated once for each control flow, and operations that would not execute in that flow are removed. This predicate-aware form of tail duplication trades schedule space for speed, and is applied by a heuristic that attempts to limit the resulting code expansion.

- **Tree traversal scheduling** [296] deals with a tree of basic blocks known as a *treeregion* [117], which is a subgraph of the execution path graph. Any path from a treeregion root to a treeregion leaf is also a superblock [87]. Consideration of a larger region provides more possibilities for ILP, and allows profile information to be incorporated dynamically rather than assuming that probabilities are fixed. The tree traversal scheduler [296] attempts to schedule branches as early as possible to minimise speculation and code duplication.

However, there are some general problems with all acyclic schedulers, as follows:

- **Loops** are not well supported. In general, trace-style scheduling algorithms require loops to be unrolled, as the *back edge* which causes the loop to repeat is not handled by acyclic algorithms. This requirement is strict in superblock, hyperblock and treeregion-based schedulers [87].

Exception Type	Caused By	Handled By
Divide by zero	Division	OS sends signal to program (e.g. <code>SIGFPE</code> on Unix) and resumes or terminates program
Memory alignment	Load/store	OS sends signal to program (e.g. <code>SIGBUS</code> on Unix) and resumes or terminates program
Page fault	Load/store	OS loads the missing memory page from disk then resumes program
TLB miss	Load/store	OS updates the <i>translation lookaside buffer</i> (TLB) [195] then resumes program

Table 6.1: Some of the exceptions that may be raised during execution of a program. The first two represent error conditions, caused by supplying incorrect parameters to an instruction. The second two are used to handle conditions that arise under normal circumstances on a computer with virtual memory.

Loop unrolling is the simplest strategy to work around this problem. Copying loop bodies increases the amount of ILP available. Loop peeling [170] is a related strategy where an exact number of loop iterations are expected, so the loop code is optimised to support that number efficiently. Both result in code expansion, and do not solve the problem of exploiting ILP across the back edge [60].

Loops are handled directly by the original trace scheduling algorithm [85], by using a book-keeping operation to handle the back edge. The back edge is a scheduling boundary, so unrolling or peeling is still important [87], and the additional complexity of handling this special case seems to have resulted in its omission from subsequent implementations [86].

- **Hazardous operations** must be handled. These operations can generate exceptions under some circumstances (Table 6.1). Exceptions may cause OS code to execute, e.g. to load memory pages from disk. It is important that exception handling leaves the program in a consistent stage so that the OS can return to it once the exception has been handled: in other words, *precise exceptions* [234] are required.

Handling hazardous operations efficiently requires hardware support. For example, [236] describes a superscalar out-of-order issue algorithm that commits the results of operations in order, including register updates and exceptions. If an operation caused an exception, it is suspended until commit time, ensuring that the order of operations is preserved. If a commit never occurs (perhaps because of a branch misprediction or another exception), the suspended exception is discarded.

Acyclic schedulers also need to ensure that exceptions are handled precisely. The simplest strategy for doing this is to ensure that instructions that can cause exceptions are never executed speculatively: [48] refers to this strategy as “restricted code percolation”, because hazardous operations are never moved across a branch point. Higher performance is possible if the CPU provides non-trapping versions of hazardous operations, where exceptions are ignored if they occur. This enables “general code percolation” [48], with the drawback that exceptions cannot be used for memory paging or to diagnose errors. To support arbitrary “boosting” percolation, the results of speculative execution must be suspended until a subsequent commit point (e.g. in a shadow register file [235]): this requires extensive additional

hardware, but allows the acyclic scheduler to ignore the possibility of exceptions.

Trace-style schedulers can be used for the generation of microcode for the TARGET RFU since they permit exploitation of ILP in general software. Any list of operations can be used. The algorithms examined in this section are efficient (near $O(n)$ for n input operations), so they meet the criteria needed for TARGET. However, looped code cannot be processed as effectively as linear code due to the scheduling boundary imposed by the back edge. The usual solution (unrolling) results in code expansion and still forces each group of unrolled iterations to be scheduled in isolation [60]. Because general programs tend to include many loops, and loops are likely to form part of a WC path, a second class of VLIW scheduling algorithms should be examined.

6.1.3 Cyclic Scheduling

Cyclic VLIW schedulers explicitly aim to optimise an inner loop, i.e. a loop that does not contain other loops. The term *software pipelining* [51] describes a general class of *cyclic scheduling* algorithms that are not subject to the back edge boundary [211], treating all control flow edges equally. The goal of a cyclic scheduler is to minimise the total time needed to execute a loop, not the total time required by any specific iteration, and to this end, cyclic schedulers overlap execution of multiple iterations in order to expose greater ILP. This process is analogous to pipelining within a CPU (section 2.2.2).

Early work on software pipelining [51] was restricted to vector instructions, where parallelism is explicit. However, work by Rau and Glaeser [212] used a *modulo scheduler* to show that the same principles were applicable to general inner loop code, because parallelism could be exposed using the data flow constraints. The input of a modulo scheduler is a data flow graph, describing an inner loop of any size. It can include any number of basic blocks, but cannot contain other loops or function calls. This loop is transformed into a *loop kernel*, which is a sequence of instructions for a VLIW machine, each instruction containing multiple operations. A loop kernel will typically contain several iterations of the original loop executing simultaneously [211]. The best schedules have the lowest *initiation intervals* (IIs): the II is the size of the kernel in clock cycles, and a measure of how well the operations could be scheduled together.

Modulo scheduling algorithms [212, 211, 165, 87] tend to start with the smallest possible II value, and attempt to generate a schedule. If this fails, the II value is incremented and scheduling is attempted again. This makes for a slow search process, which is slowed further by the use of backtracking in many (but not all) algorithms. Modulo scheduling is a complex problem because data dependences must be considered in both directions: past and future. In contrast, trace scheduling considers data dependences in one direction only.

Llosa et al. proposed *swing modulo scheduling* (SMS) [165], which does not involve any backtracking and compares well to other modulo scheduling approaches [54]. However, it is still necessary to search all possible II values in the hope of finding the smallest one that produces a working schedule. Additionally, modulo scheduling is only suitable for one specific type of code: a loop that contains no other loops. Because of this, modulo scheduling does not meet the requirements of the WCET reduction process (section 3.2).

6.1.4 Using VLIW Scheduling Algorithms for TARGET

Both classes of VLIW scheduler identified in [87] have been examined. Conventional implementations of these algorithms aim to reduce program ACETs, but adaptations for WCET reduction are possible. Acyclic schedulers are conventionally guided by profile information, but WC path information from a WCET analysis process such as IPET could be used instead. The general problem is that WCET reductions interact with each other in a way that ACET reductions do not, creating a new WC path as soon as the previous WC path has been reduced [80]. However, VLIW-style scheduling has been previously applied to reduce program WCETs. Zhao [295] describes a compiler that forms superblocks as part of code generation. The process is driven by WC path information evaluated at the function level.

Of the two classes of algorithm, acyclic “trace-style” schedulers are most suitable for TARGET, because (1) trace-style schedulers can exploit ILP in any code, and (2) trace-style schedulers are very efficient: backtracking is not used. Cyclic “modulo” schedulers are only suitable for inner loops, not general code, and their algorithms always involve an iterative search step to find the smallest minimum initiation interval.

Therefore, MCGREP-2 will make use of trace-style scheduling. The simplest trace-style algorithm is superblock scheduling [48], which also forms a natural base for hyperblock scheduling (superblocks with predication) and treeregion scheduling (multiple superblocks arranged as a tree) so future improvements to the scheduler are enabled by this choice. The similarities between the VLIW code generation process and user microprogramming indicate that superblock scheduling will be suitable for RFU microcode generation in the arrangement proposed for TARGET (section 4.2.2).

But the possibility of future support for modulo scheduling will also be considered, since further WCET reductions can be obtained using modulo scheduling for the special case of looped code. Writing about VLIW compilation in general, Fisher states that [87]:

“Current practice embraces both approaches, and product compilers include both cyclic and acyclic schedulers... the two techniques are viewed as complementary rather than competitive.”

In order to support this, a “plug-in” model should be used for the scheduler. The RFU elements should also support predicated operation, to support the future implementation of both modulo scheduling and hyperblock scheduling.

6.2 MCGREP-2 TARGET Generator

MCGREP-2 inherits features from MCGREP-1, but the design is improved in accordance with the aims stated at the beginning of this chapter:

- Support for **automatic microcode generation**, from multiple types of scheduler. Two features are needed to support this - firstly, a configurable CGRA-like component that matches the needs of a particular type of scheduler, and secondly a microprogramming interface to translate symbolic microprogramming commands into encoded microinstructions (section 6.2.4). The interface will allow any compatible microcode generator to be connected to any version of MCGREP-2. A WCET reduction process (Figure 6.1) can make use of the automatic generator components.

Parameter	Configurable?	Notes
ALU function set	yes	Part of architecture definition - can be subclassed. (Py)
Interconnect	no	
ISA	yes	Set by the architecture definition. (Py)
l, m, n values	yes	The value of $l + m + n$ (see Figure 4.5) can be controlled by <code>num_units</code> in an MCGREP-2 configuration parameter file. (Cfg)
Register file dimensions	yes	Controlled by block RAM driver. Two block RAM drivers are implemented: RAMB16 [290] and RAMB4 [283]. (Cfg)
Built-in microprogram	yes	Set by the architecture definition. (Py)
Microcode interface	yes	Automatically generated, as in section 5.2.1. (Auto)
μ code store dimensions	yes	Controlled by block RAM driver, like the register file dimensions. (Auto)

Table 6.2: Parameters for MCGREP-2 generator implementation (see section 6.2.1). Parameters marked with (Py) can be adjusted by extending MCGREP-2 with Python [208] code. Those marked with (Cfg) can be adjusted in the configuration parameter file. Those marked with (Auto) are automatically adjusted to match other parameters.

- **Better parameterisation support:** this is provided by improved code modularisation, which allows greater configurability through MCGREP-2 extensions (section 6.2.1).
- **Faster simulation:** this is possible using a JIT-like approach (section 6.2.5).

The components that are improved in MCGREP-2 are discussed in the following sections. Sections 6.2.1 to 6.2.3 describe extensions to the microarchitecture generated by MCGREP-1. Section 6.2.4 describes the microprogramming interface and section 6.2.5 describes the new simulator. After this, section 6.3 describes the implementation of an automatic microcode generation tool based on a superblock scheduler.

6.2.1 Features

The parameterisable features of MCGREP-2 are summarised in Table 6.2. (The equivalent table for MCGREP-1 is Table 5.5.) This section examines each feature in detail.

- **Timing Properties:**

The basic block timing invariance property of MCGREP-1 CPUs (demonstrated in section 5.3.5) is retained for MCGREP-2 CPUs. Each instruction (or microinstruction) takes an exact number of clock cycles provided that memory responds deterministically. This feature is required for TARGET.

However, in response to the need for a higher maximum clock frequency (identified in section 5.3.7), MCGREP-2's microarchitecture is redesigned to include a simple pipeline. The pipeline decodes the next instruction "early", so that instruction decode and dispatch are not part of the same combinational path. As the ORBIS32 ISA has only delayed branches [195], no branch penalty is introduced by this.

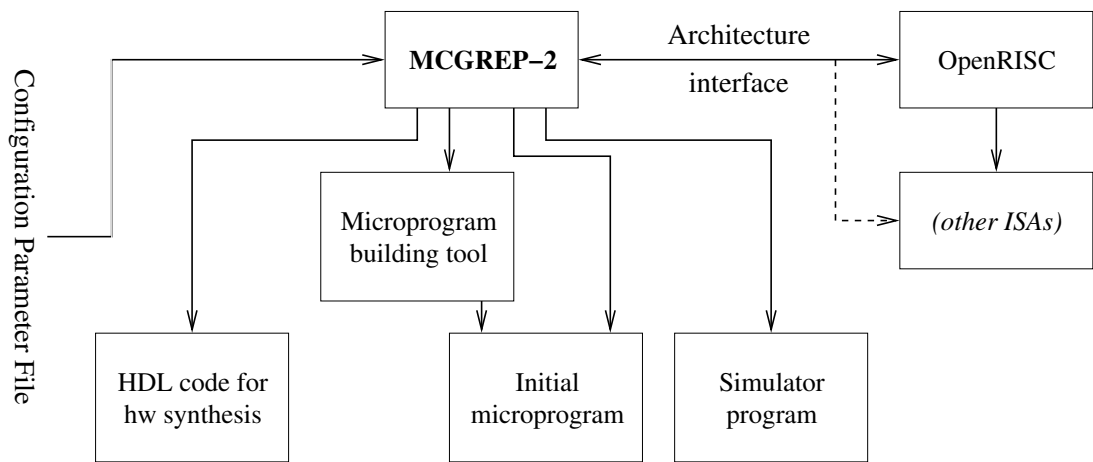


Figure 6.7: The role of the architecture interface. MCGREP-2 generates the components required for an implementation of TARGET (Figure 4.5) based on two inputs. One is a configuration parameter file, which allows simple constants to be defined, and the other is an architecture, which is an extensible code module defined in Python. The architecture allows MCGREP-2 to be parameterised in many ways (Table 6.2), and new architectures may extend the default OpenRISC architecture module.

- **Microprogram Interface and Store Dimensions:**

As in MCGREP-1, all four of the components required for a model of TARGET, i.e.

- the simulator,
- the initial microprogram generator,
- the HDL model,
- and the user microprogram generator,

are built from a common parameterisable core that reads (and writes) microprograms. The language definition is controlled by a codec, as used in MCGREP-1 and described in section 5.2.1. The codec is automatically generated based on other parameter settings, and (as in MCGREP-1) the microprogram generation mechanism is able to adjust the size of the microprogram store as appropriate.

- **ISA and Built-in Microprogram:**

In MCGREP-1, the built-in microprogram was generated by an abstraction layer (Figure 5.6), but this was not exposed for user modification. The obvious improvement is to allow extensions to be made to this microprogram via object-oriented inheritance. In MCGREP-2, parameterisable features (Table 6.2) are enhanced by the abstraction of many ISA-dependent functions (and ISA-dependent hardware) into an extensible module called an “architecture” (Figure 6.7). Architectures can inherit from each other, allowing ASIP-style extensions to be incorporated into CPUs generated by MCGREP-2 and permitting “ports” to new ISAs.

The ISA and built-in microprogram are two of the features that can be extended in this way. The following components are provided by the architecture interface (Figure 6.7):

uPC	Reg. File Settings				ALU Op	ALU In		Effects
	Port A	Port B	Port C	Port D		0	1	
branch	Rd r0	Wr pc	Wr pc	Rd r0	ARITH_ADD	C	lsu	
sys	Rd r0	Wr pc	Wr pc	Rd r0	ARITH_ADD	C	lsu	Dispatch(nop) LSU(ifetch)
nop	Rd r0	Rd r0	Rd pc	Rd r0	PASS	A	B	
sys	Rd r5	Wr pc	Wr pc	Rd r0	ARITH_ADD	C	lsu	Dispatch(slli) LSU(ifetch)
slli	Wr r6	Rd r0	Rd pc	Wr r6	SHIFT_LEFT	A	lsu	
sys	Rd r6	Wr pc	Wr pc	Rd r0	ARITH_ADD	C	lsu	Dispatch(srai) LSU(ifetch)
srai	Wr r6	Rd r0	Rd pc	Wr r6	SHIFT_RIGHT	A	lsu	
sys	Rd r8	Wr pc	Wr pc	Rd r0	ARITH_ADD	C	lsu	Dispatch(load0) LSU(ifetch)
load0	Rd r0	Rd r0	Wr ea	Rd r0	ARITH_ADD	A	lsu	
load1	Rd r0	Rd r0	Rd r0	Rd r0	PASS	C	B	LSU(load word)

Figure 6.8: A sample microexecution trace as executed by one unit when acting as an interpreter for ORBIS32 instructions.

- Python methods for specifying opcodes, and generating microinstructions to handle them, like those shown in Figure 5.6.
- A dispatch unit generator to automatically produce an opcode decoding tree.

These features are not tied to any particular ISA, but an implementation of the ORBIS32 has been written using them. Although the Microblaze ISA was found to make more efficient use of code space with the available C compiler (based on instruction counts for benchmark code, Table 5.12), ORBIS32 is still a good choice of ISA for the reasons listed in section 5.2.2. It is a simple ISA (Table 5.3), with reference HDL and simulator, and the MCGREP-1 implementation can be translated to MCGREP-2, saving time.

For MCGREP-2, the set of special microprograms is updated as shown in Table 6.3. A new addition is the System microprogram. As in MCGREP-1, a cyclic execution model is used (section 4.2.1), but microprogram states form every part of the cycle in CPUs generated by MCGREP-2 to avoid extending the control unit with a large number of microbranch types, as seen in Table 5.8. When an instruction completes, it always returns to the System microprogram, which performs housekeeping operations and jumps to the handler for the next instruction (Figure 6.9). A typical trace of microprogram execution is shown in Figure 6.8. Useful properties of the approach include: (1) the ALU is reused to increment the program counter, so a separate adder is not required, (2) fetch and decode happen in parallel with execution, and (3) more operations of the control unit are placed under microprogram control.

- *l, m, n:*

MCGREP-2 allows control of these TARGET parameters (see Figure 4.5) through a simple configuration mechanism. The total value of $l+m+n$ is controlled by the `num_units` setting in an MCGREP-2 configuration parameter file. This allows the array size to be expanded (or reduced) easily. The default architecture configuration forces $n = 1$ and $m = 0$, i.e. exactly one unit may access the memory and execute machine code. However, the array can include

Name	Purpose
Boot	Initialises the CPU.
System	Decodes the next instruction, uses a dispatch table to jump to the appropriate address, and begins fetching the next instruction (Figure 6.9).
IllegalEx	Handles illegal instructions by jumping to address 0x700 as specified by the OpenRISC manual [151].

Table 6.3: Special microprograms for CPUs generated by MCGREP-2.

```
def System_Cycle():
    # nextPC <- curPC + 4
    self.Unit_Program(MUX_ODD, MUX_FOUR, ARITH_ADD)
    self.Bank_Port_Program(1, 0, PROGRAM_COUNTER, True)
    self.Bank_Port_Program(0, 1, PROGRAM_COUNTER, True)
    # Fetch instruction at [curPC]
    self.LSU_Prepere(False, 4, False, True)
    # Load registers AS and BS
    self.Bank_Port_Program(0, 0, FROM_IWORD_A, False)
    self.Bank_Port_Program(1, 1, FROM_IWORD_B, False)
    # Next UPC is taken from the instruction decoder.
    self.UPC_Dispatch()
```

Figure 6.9: The System microprogram, which is executed between every instruction. System carries out the following actions: (a) starts fetch of instruction $i + 1$, (b) increments the program counter, (c) sets up registers for instruction i , and (d) dispatches instruction i , which begins executing in the next clock cycle.

any number of additional units capable of arithmetic and logical operations, so $l \geq 0$. The reason for this design is that accesses to instructions or data in memory are subject to the memory bottleneck, so there is no benefit to parallel access without a more advanced memory architecture. The design is an improvement on MCGREP-1 which did not permit adjustment of the array size parameters.

- **Interconnect:**

The MCGREP-2 CPU generator assumes a fully connected CGRA interconnect in the default architecture. This will not scale to support any number of RFU elements, but that problem is offset by the limitation of ILP in general software [268]. Restrictions to the network could be implemented as architectural extensions at a later date, but because the ILP limitation bounds the useful array size at around 4 units, there is no need to implement these components as part of MCGREP-2.

- **ALU function set:**

The MCGREP-2 ALU function set can be extended as part of the architecture. A basic set of ALU functions is provided for the ORBIS32 ISA: this is the same as the MCGREP-1 set, listed in Table 5.7. However, new symbols can be defined and implemented. Such extensions must be written in two languages, which are embedded in the MCGREP-2 Python code:

- VHDL, for inclusion in hardware descriptions generated by MCGREP-2 (sections 6.2.2

MCGREP-2 Features

▷ /mcgprep2-src/mcgrep/architectures

MCGREP-2 architecture definitions.

▷ /mcgprep2-src/mcgrep/architectures/openrisc

ORBIS32 ISA definition.

▷ /mcgprep2-src/mcgrep/simulator/Simulator.py

Simulator generator for MCGREP-2.

▷ /mcgprep2-src/mcgrep/generator/mcgprep2vhdl.py

VHDL generator for MCGREP-2.

▷ /mcgprep2-src/mcgrep/block_ram.py

Block RAM generator: used for microprogram store and register files.



and 6.2.3);

- C, to form part of the simulator or microprogramming interface (sections 6.2.4 and 6.2.5).

Optionally, the extended features can be added to the ISA, either by extension of the built-in microprogram, or by the creation of a new microprogram. They can also be supported by microcode generating software, which is discussed in subsequent sections.

- **Register file dimensions:**

As in MCGREP-1, the register file dimensions are set by the block RAM driver. Two drivers are available, and may be selected in the configuration parameter file. The RAMB4 driver is used for Spartan-2E and earlier FPGAs, and provides 4Kbits of memory per block RAM [283]. The RAMB16 driver is used for more recent FPGAs such as Spartan-3, Virtex-II, and subsequent devices, and provides up to 18Kbits of memory per block RAM [290]. Currently, there is no support for non-Xilinx FPGAs. However, the memory driver subsystem could be extended to support other types of embedded RAM.

6.2.2 Microarchitecture - Top Level

Like its predecessor, MCGREP-2 uses VHDL templates to produce HDL code to implement an instance of TARGET. As the array size is configurable in MCGREP-2, a two-level structure is used for the generated VHDL. The lower level (“MCGREP-2 unit”) is described in section 6.2.3. Figure 6.10 shows the top level of this structure, which implements the CPU core. The components in the top level are (from top to bottom, then left to right):

1. **Interconnect:**

In CPUs generated by MCGREP-2, the fully connected communications network permits any even-numbered ALU port to read from any even-numbered register file port. An identical network provides the same feature for odd-numbered ports. This layout is based on the observation that operation inputs are generally tied to a particular register file port.

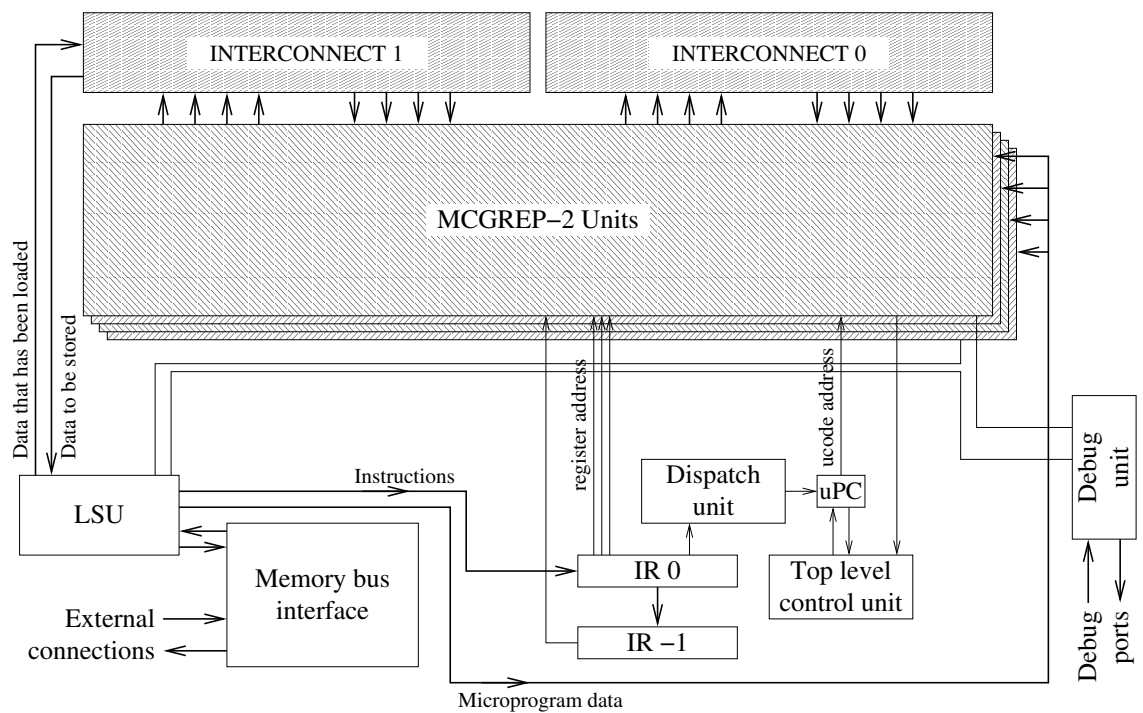


Figure 6.10: Top level of CPU generated by MCGREP-2, containing all of the components that are shared between every unit. The internals of each unit are shown in Figure 6.11.

For example, consider the ORBIS32 subtract operation `l.sub r3, r4, r5`. The correct result ($r3 \leftarrow r4 - r5$) is dependent on supply of $r4$ to ALU port 0, and supply of $r5$ to ALU port 1. The OpenRISC CPU microarchitecture arranges for $r4$ to be loaded on register file port 0, and $r5$ to be loaded on register file port 1, then directly links register file ports to ALU ports.

In ORBIS32, there is never any need to exchange the inputs such that (for example) register file port 1 is connected to ALU port 0. This is a feature of the ISA that simplifies the CPU hardware required to implement it. MCGREP-2 takes advantage of this property to simplify the interconnection network, which connects (at most) half the register file ports to half the ALU ports across many units.

2. LSU:

The MCGREP-2 load-store unit (LSU) is based on the MCGREP-1 LSU, as described in section 5.2.3.1. As in CPUs generated by MCGREP-1, only one of the functional units can access memory (in Figure 4.5, $n = 1$ and $m = 0$). The MCGREP-2 LSU has full support for byte steering, because supporting this feature using a partly microprogrammed approach proved to be costly in several ways:

- Additional relative microbranch instructions were needed specifically to support the byte steering feature (`UCTRL_JR_EA` in Table 5.8).
- Load and store operations required more microprogram space (e.g. 6 microoperations per load operation, Figure 5.9).
- Otherwise unused sign extension commands were required (e.g. `SIGN_EXTEND_24`).

Symbol	Function
UPC_BRANCH	Absolute jump to a uPC value provided by a register.
UPC_BRANCH_IF	Jump if a predicate is true, otherwise step (see below).
UPC_BRANCH_IF_NOT	Jump if a predicate is false, otherwise step.
UPC_DISPATCH	uPC set to output of dispatch unit. Used at the end of the System cycle.
UPC_STALLED	Pause in the current uPC state, with register updates inhibited, until the current memory access (if any) completes. At that point, allow register updates and step.
UPC_STEP	$uPC \leftarrow uPC + 1$
UPC_SYSTEM_CYCLE	Jump to System cycle (Figure 6.9). Used at the end of each instruction handler.

Table 6.4: The set of uPC branch commands supported by MCGREP-2 is smaller than the set for MCGREP-1 (Figure 5.8).

Placing these features in the LSU rather than an ALU ensures that they can be used by all units that are capable of accessing memory, and improves the efficiency of the microprogram.

3. Memory Bus Interface:

The MCGREP-1 memory bus interface (section 5.2.3.3) has been abstracted in MCGREP-2 as a “bus driver”, which allows different bus standards to be used by implementation of an appropriate driver.

4. Instruction Registers and Dispatch Unit:

These components are nearly identical to their counterparts in the MCGREP-1 design (section 5.2.3.8). The layout of the decoding tree used by the dispatch unit is similar to Figure 5.9, with differences only in the microprogram addresses at the leaves. The VHDL implementation has the same overall structure in both cases. However, the MCGREP-2 microarchitecture places instruction register and dispatch components on the top level. This forces $n = 1$ (see Figure 4.5) and prevents SMT (section 2.2.2.6), but SMT is limited in any case by the effects of the memory bottleneck. The arrangement is a natural consequence of only being able to access one word of memory at a time.

5. uPC and Top Level Control Logic:

As in MCGREP-1, a single uPC register is shared by all array units. This avoids the need to explicitly synchronise the operations of different parts of the array. The top level control logic manages the uPC register according to instructions found within the microprogram for the first array unit. The supported operations are listed in Table 6.4.

A new feature of MCGREP-2 is support for conditional operations using predicates. Two such conditional operations are `UPC_BRANCH_IF` and `UPC_BRANCH_IF_NOT`. The condition for these uPC operations is taken from the predicate input of the ALU for the first array unit.

6. Debugging Features:

Hardware generated by MCGREP-1 was tested and debugged using interactive software running on an FPGA (section 5.3.1). While this approach works, the FPGA-based debugging

software (Figure 5.16) is not user friendly and does not permit automatic testing. Thus, testing and debugging are time consuming.

In order to address this problem, MCGREP-2 generates a debugging chain similar to a JTAG bus [193]. The JTAG standard was originally intended for testing inter-IC connections on circuit boards after manufacturing, but it has found additional applications including FPGA configuration (section 2.5.2) and debugging complex devices such as CPUs. JTAG enables access to the internal registers of a device using a *scan chain*, which is effectively a very wide shift register. On receipt of one JTAG command, the scan chain is loaded from the internal registers. On receipt of a second command, the chain is shifted by one bit, producing one of the internal bits on the JTAG bus output. This second command can be repeated as many times as necessary to obtain the value of the entire chain.

One CPU that uses JTAG for debugging purposes is OpenRISC [151], but JTAG-like buses are often found in CPUs. For example, a JTAG-like serial debug chain was included in early RISC CPUs [144], and helped researchers to find flaws in the hardware. For this purpose, a serial bus is superior to a parallel bus for providing access to internal registers because hardware costs are lower.

MCGREP-2 reuses the principle of a serial debug chain, but adopts a custom bus in place of JTAG to simplify integration into a test harness. A standard RS232 serial connection can be used to send the following commands to the hardware test controller:

- CPU clock step by n cycles, with $0 < n < 65536$,
- Download current debug chain value from CPU,
- Set CPU system clock to “free run”,
- Set memory latency to M cycles,
- Reset CPU.

Additionally, the test controller can act as a “pass through”, relaying serial information between the CPU hardware and a workstation. This is useful for uploading programs onto the hardware. The capabilities are similar to those provided by the Microblaze “Microprocessor Debug Module” [286].

Software on the workstation is able to decode a debug chain value to retrieve the internal register values. This enables inspection of CPU registers and control lines within both the LSU and the functional units. Combined with the facility to single-step the processor, this is a powerful debugging tool.

The test controller is implemented as a VHDL state machine. An earlier implementation was derived from the harness used for testing MCGREP-1 hardware (Figure 5.15). The commands listed above were used in place of an interactive interface, and the implementation was highly flexible because changes could be made to the test controller’s embedded software without needing to rebuild the FPGA bit file. But the T80 [269] soft core, bus and RAM used up a large area of the FPGA, so after the feature set was finalised, the functionality was ported to a pure VHDL design to leave more space for experiments with MCGREP-2.

7. Microprogram Distribution Network:

Top Level

▷ /mcgrep2-src/mcgrep/generator/mcgrep2vhdl.py

VHDL generator for MCGREP-2, includes LSU, microprogram distribution network and memory bus interface source code.

▷ /mcgrep2-src/scripts/mcgrep_hardware_test.py

Debugging tool: uses the debug chain to compare operation of MCGREP-2 CPUs on an FPGA with a simulator on a host PC.

▷ /mcgrep2-src/vhdl

Test and debugging harness VHDL code.

▷ /mcgrep2-src/scripts/mcgrep_make_vhdl.py

VHDL generator for MCGREP-2 CPUs.



This component maps part of the memory space to the microprogram memory of each unit. In CPUs generated by MCGREP-1, the network only allowed a single microprogram memory space to be addressed, shared between all units. Programs updated that memory by writing to a specific address X within a reserved area of the program memory space (e.g. between $0x10000000$ and $0x1ffffffc$). X contained encoded values of the row and column address of the RAM cell to be updated.

CPUs generated by MCGREP-2 have one microprogram memory space per functional unit, so the update address X must also include a unit number. Microprogram updates are distributed by a grid of data connections to each unit. There is also a grid of “update enable” lines to activate specific destination units for each write.

6.2.3 Microarchitecture - Unit Level

The microarchitecture of each MCGREP-2 unit is based on the MCGREP-1 design (Figure 5.7), but with some changes aimed at avoiding the effects of the register port limitation described in section 5.3.7). Figure 6.11 shows a single unit. Before the components of Figure 6.11 can be described, a new design feature should be pointed out.

In CPUs generated by MCGREP-1, the simple strategy of requiring 2 clock cycles per microinstruction was used to avoid the register port bottleneck imposed by FPGA block RAM. A consequence of this is that each ALU is used at most 50% of the time.

The register port limitation still applies in CPUs generated by MCGREP-2, but it is addressed by two strategies that allow the ALU to be continuously active. The first strategy is mirroring: the register files within a single array unit are copies of each other, at least as far as general purpose registers are concerned. This allows reads to be performed on either unit. The second strategy involves interleaving the two types of operations carried out by the CPU, specifically the house-keeping operations of the System cycle (Figure 6.9) and the code execution operations of other microinstructions.

MCGREP-2 limits each microinstruction to the register accesses that are possible within one cycle. The System cycle prefetches register values that are likely to be used by the next cycle,

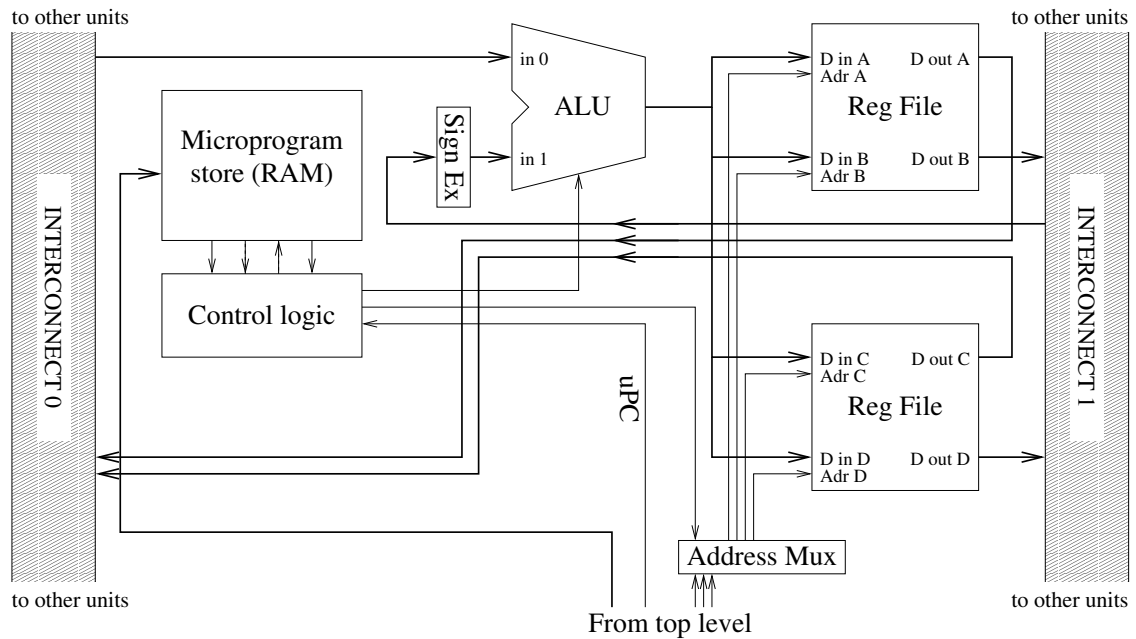


Figure 6.11: Architecture diagram for each MCGREP-2 unit within Figure 6.10.

specifically those indicated within the *rA* and *rB* fields of the next instruction word (Figure 5.8), so that instruction handlers do not normally need to fetch register values explicitly.

The effect of this is illustrated by the microexecution trace in Figure 6.8. Note that each line represents a single clock cycle. The ALU is almost always in use and most instructions complete in two clock cycles. Additionally, as there is exactly one clock cycle per microinstruction, the control logic is simplified. The components of Figure 6.11 are, from left to right:

1. Microprogram RAM and Control Logic:

The purpose and arrangement of these components is similar to that in MCGREP-1 (section 5.2.3.10). However, one change is made. In MCGREP-2, the control logic maintains a 1-1 mapping between microoperations and clock cycles. Unless a microoperation gives a specific *uPC* branch command (Figure 6.4), each microoperation takes one clock cycle.

2. Sign Extender and ALU:

These components are mostly identical to their counterpart in MCGREP-1 (section 5.2.3.4). The basic set of supported microoperations are shown in Table 5.7. Three significant changes are:

- The VHDL is now generated by an extensible architecture module in place of a fixed template, allowing ALU features to be extended.
- The sign extender now only supports the features required by ORBIS32 (Table 5.6) as the features needed to carry out byte steering in the ALU (e.g. `SIGN_EXTEND_8`) are no longer required.
- All ALU functions can be conditional. A predicate input can be assigned to any microinstruction using the ALU. If the predicate is true when the microinstruction executes, the

results of the ALU operation are committed to the register file. Otherwise, they are discarded.

Predicates are stored in the same register banks as general data. They are required by certain types of microcode generator, such as those based on hyperblock [170] and modulo [211] schedulers. Predicates are also useful for acyclic scheduling in general, because they allow condition flags to be handled using the same mechanism used to rename and resolve dependences between other registers.

3. Address Mux:

In MCGREP-1, the address mux controls the register file address ports so that microprograms can use the rA, rB and rD instruction word fields (Figure 5.8). In MCGREP-2, the same arrangement is used, although instruction word information is now obtained from the top level (Figure 6.10).

4. Register Files:

As for MCGREP-1, two banks of block RAM are used to provide register files (section 5.2.3.7). The difference in implementation is seen in the surroundings. Because block RAMs only have two ports, it is not possible to simultaneously read two values and write two others. So the same ALU output is mapped to every block RAM input, allowing a microprogram to:

- On each bank, write to either one of the input ports in order to reserve the other for a read operation,
- And either: treat the two banks as coherent, by ensuring that every write to one bank is also carried out on the other,
- Or: treat the two banks as separate, doubling register space.

These options give microprograms a variety of flexible ways to use the hardware. Up to four reads can be performed on a single unit. If a coherent write is required, then two reads can be performed. If a single bank write is required, then three reads can be performed. The use of separate banks does provide increased space, but the cost is reduced register access bandwidth. Each microprogram can make this tradeoff as required.

The built-in microprogram for the ORBIS32 ISA ensures that general-purpose register values and the program counter register are coherent across each bank. The register files mirror each other as described in section 6.2.3: every update in Figure 6.8 involves a write on ports A and D. This coherence even makes it possible to remove the need for a System microprogram (Figure 6.9) and execute one instruction per clock cycle, but this would require additional hardware to compute program counter addresses and pipeline memory accesses, and the memory bottleneck limitation would still apply.

6.2.4 Microarchitecture - Programming Interface

MCGREP-2 must support automatic microcode generation using a variety of schedulers, with a plug-in model to allow schedulers to be changed easily. This requires a microprogramming interface that is usable from any program.

In MCGREP-1, microprograms were always built from Python code by calling generator methods. In MCGREP-2, this is still possible, and indeed this feature is used for generation of the built-in

Unit Level Components

▷ /mcg2-src/mcg2/generator/mcg2vhdl.py

VHDL generator for MCGREP-2.

▷ /mcg2-src/mcg2/architectures/openrisc

Definitions of ALU functions and microprogram symbols for OR-BIS32 ISA.



```

/* load value */
MCGREP2_Bank_Port_Program ( & now [ 0 ] , 0 , 1 , gpr , False ) ;
Enstack_Microcode ( me , now ) ;
/* store value */
for ( bank = 0 ; bank < ( NUM_UNITS * 2 ) ; bank ++ )
{
    unsigned unit = ( bank / 2 ) + first_unit ;
    MCGREP2_Unit_Program ( & now [ unit ] ,
                          MUX_ODD , 0 , ALU_PASS_OPERATION ) ;
    MCGREP2_Bank_Port_Program ( & now [ unit ] ,
                              ( bank % 2 ) , 0 , target , True ) ;
}

```


Figure 6.12: Microprogramming example: this C code generates two microinstructions that copies register `gpr` from bank 0 to register `target` on every register bank. The `Enstack_Microcode` procedure advances to the next microinstruction. Microinstruction data is stored within the variable named `now`.

microprogram. But it is augmented by an additional C-based interface, which allows C programs to generate microprograms. This allows any scheduler or code generator to be connected to the MCGREP-2 microprogramming “back end”, and is a natural extension to the MCGREP-1 feature which embedded an Upload procedure within programs (section 5.2.4).

The new C interface includes a header file which specifies the fundamental constants of any specific hardware generated by MCGREP-2. It also provides declarations for a group of microprogramming functions which enable microinstructions to be generated from RTL-style code: a usage example appears in Figure 6.12. The definitions of these functions are also generated automatically and can be incorporated into any program using `#include` or linking. Declarations stay the same for every CPU generated by MCGREP-2, although the implementations change to accommodate alterations to the parameters.

An interesting side effect of the introduction of a complete C interface for microprogramming is that programs running on the CPU can generate and upload microprograms without external intervention. Thus, run-time specialisation and JIT compilation are possible. These are not desirable in a hard real-time context because of the difficulty of predicting the outcome of specialisation or JIT compilation, but may be useful in situations where non real-time tasks are also in use.

C Microprogramming Interface



▷ /mcg2-src/mcgrep/generator/mcgrep2cmicrogen.py
Interface generator for MCGREP-2 CPUs.

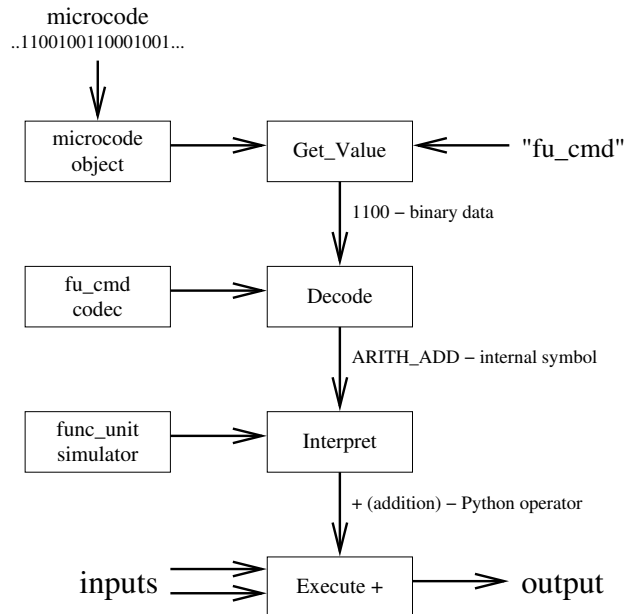


Figure 6.13: Steps taken to decode an ALU command from the internal representation of microcode to an actual machine operation (in this case, addition).

6.2.5 Simulator

MCGREP-1 used a Python-based microprogram interpreter as a simulator (section 5.2.5). This was a performance simulator in the sense described in section 5.1, with cycle-accurate simulation of the MCGREP-1 hardware, which is required to correctly represent microprogram operation. However, instruction throughput was poor (section 5.3.7). This is because of the overhead of the simulator: an interpreted Python program that interprets microprograms, which may themselves interpret the ORBIS32 ISA. These levels are illustrated in Figure 6.13.

Improved performance requires the removal of some levels of interpretation: *vertical migration* of MCGREP-2 microprograms into the native machine code of the simulator host (e.g. x86). The following strategies could be used:

- **Automatic vertical migration** using a JIT compiler.

This performance improvement strategy is easy to implement in Python using the Pyco JIT compiler [216]. However, Pyco was only found to introduce a 2-3 times improvement when applied to the MCGREP-1 simulator, probably due to the inherent difficulty of automatically optimising the data dependent steps in Figure 6.13.

- **Converting microprograms to Python code.**

This strategy removes a layer of interpretation, as microoperations do not need to be decoded dynamically. Rather, microoperations are decoded the first time that a particular uPC value is reached, a process which is repeated only if the microprogram at that uPC address is changed.

However, experiments with the MCGREP-1 simulator suggested that the total speedup would still be less than one order of magnitude, suggesting that microprogram decoding is only one of many factors that slow the MCGREP-1 simulator.

- **Converting microprograms to native code.**

This strategy removes all interpretation, replacing a microprogram with equivalent native machine code for the simulator host. Direct conversion to native code is not feasible due to (1) the complexity of code generation, and (2) the undesirable dependence on a single platform. However, conversion is possible via an intermediate language that compiles to native code, such as C. This means that a C compiler will be required on the simulator host, but `gcc` is almost ubiquitous on present-generation Unix workstations.

The microprogram can be changed, so it must be possible to repeat the conversion process if an update occurs.

The best strategy was deemed to be the one with the greatest potential for performance improvement while retaining cycle accuracy: conversion to native code. However, another simulator requirement is flexibility (Figure 5.3). The MCGREP-1 simulator was easy to extend due to the pure Python implementation, and extensions were used for testing (Figure 5.14). A hybrid C/Python program introduces special difficulties for extension, as both C and Python code may need to be extended, and the correct interface may need to be maintained between them.

The MCGREP-1 interpreting simulator is replaced with a code generator (Figure 6.14, label “1”). This translates microprogram data into C source code, removing layers of interpretation. For example, Figure 6.13 is collapsed to a single line of C code to perform an addition. The result of translation is compiled and used for simulation (Figure 6.14, label “2”).

This approach avoids the performance problems of MCGREP-1, as there is no online interpretation. It also supports changes to the microprogram that are made at runtime: these force a rebuild of the simulator core (“2”), which stores all machine context information in a separate object (“5”) to allow the machine state to survive this process. The update process is handled by the simulation interface (“3”), a C program that provides an interface between the controller, implemented in Python, and the C simulator core. The controller forces a rebuild whenever an attempt is made to execute a microprogram that has been updated. Rebuilds can take some time, but the benefit is faster execution.

The simulator core (“2”) presents a number of external interfaces for communication with the simulation harness (“3” and “4”). One is a Run function. This continuously executes a simulation until (1) a microcode update is required, (2) the program terminates, or (3) an error occurs. Until one of these three events occurs, execution takes place entirely within the Run function or its callees. There are no calls to Python functions unless *hooks* have been installed.

Hooks act as logic probes for testing. Since the simulator core is C, rather than Python, it cannot be arbitrarily extended. Features can be added by extending the code generator, but this would not be suitable for extensions that interacted with other program components during execution, as these would need to cross the C/Python language barrier, and it would be difficult to support this without introducing a complex namespace dependence between code in the MCGREP-2 software and code

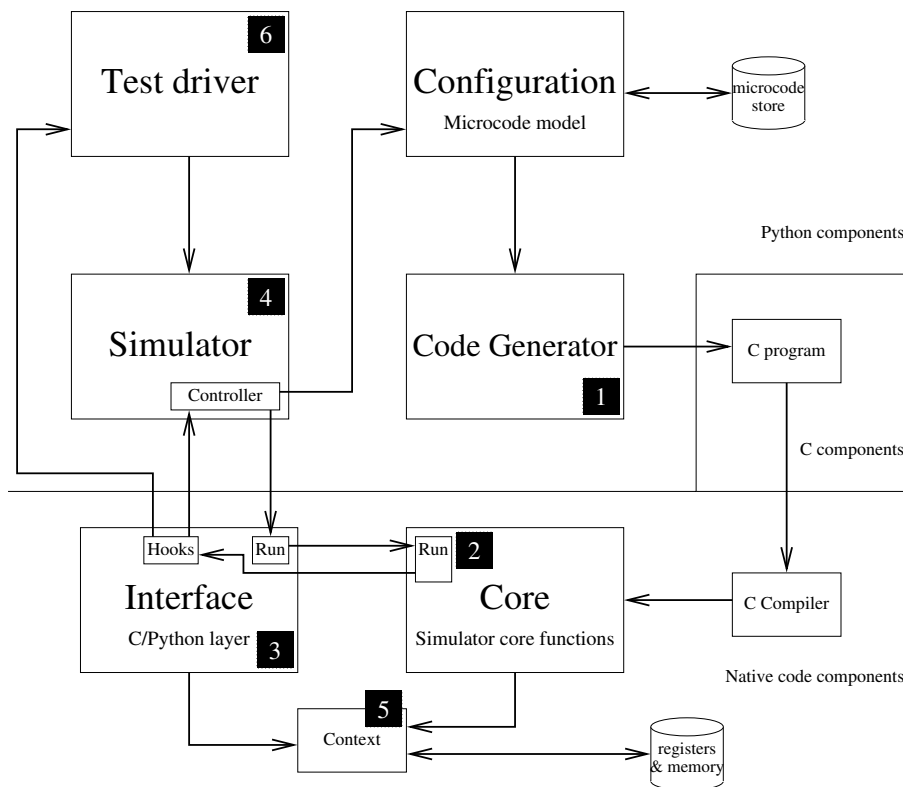


Figure 6.14: Architecture of MCGREP-2 simulator.

Simulator

- ▷ /mcgrep2-src/scripts/mcgrep_simulator.py
MCGREP-2 simulator program.
- ▷ /mcgrep2-src/mcgrep/simulator/Simulator.py
Simulator source code.
- ▷ /mcgrep2-src/mcgrep/simulator/SimulatorInterface.c
Source code for C/Python interface.

elsewhere. So hooks are provided to formalise the extension process, through the interface listed in Table 6.5. The functions registered with this interface are called by the simulator in response to specific events.

Table 6.5 lists the hooks available in the current version and the events that cause them. All of these are optional and are deactivated by default. The core provides an interface to allow hook functions to be registered, which can be used by testing and debugging code (“6”).

6.3 MCGREP-2 Trace-style Scheduler

An automatic microprogram generator has been identified as a requirement for each implementation of TARGET, including CPUs generated by MCGREP-2. The modular design of the MCGREP-2 software supports multiple microcode generators: each must make use of the interface described

Name	Description
HOOK_DISPATCH_PRE	Called whenever the dispatch table (for interpreting machine code) is accessed.
HOOK_TICK	Called once per simulated clock cycle.
HOOK_DESTROY	Called when a context object is being garbage collected by Python.
HOOK_ENTER_UC	Called in the event of a “branch to microprogram” instruction (e.g. Figure 5.10).
HOOK_UPC_BRANCH	Called when a microbranch operation is executed. (These set the microprogram counter uPC to a register value.)
HOOK_NOP	Called when an extended nop (no operation) instruction is executed. The OpenRISC simulator <code>orlksim</code> uses these to receive simulator commands from programs that are ignored by OpenRISC hardware, such as <code>exit(1.nop 0x1)</code> .

Table 6.5: Types of hook provided by the MCGREP-2 simulator.

in section 6.2.4 in order to encode symbolic microoperations as part of a microprogram. Section 6.1.4 identified a superblock-based algorithm as the best fit for the requirements, which demand a microcode generator that (1) can use any machine code as input, and (2) will execute efficiently.

To enable WC path optimisations to be applied to any program as shown in Figure 6.1, independent of the source language or the task being carried out, the trace-style scheduler takes only the following inputs:

- **P:** a program in binary form, as emitted by a compiler and linker;
- **A:** the starting point of a WC path fragment in the form of a memory address;
- **Path Function:** a function that takes one input (the memory address of a branch operation) and returns one of the following symbols depending on the action to be taken at that address: (1) `BR_ASSUME_TAKEN`, (2) `BR_ASSUME_NOT_TAKEN`, (3) `BR_ALWAYS_TAKEN`, (4) `BR_NEVER_TAKEN`, or (5) `BR_END_TRACE_NOW`;
- **B:** (optional) the memory address of a breakpoint where tracing should stop, if the stopping point is not a branch.

In section 6.1.2, five distinct steps were identified as taking place in each application of a trace-style scheduling algorithm. The fifth step, “repeat as necessary”, is only applicable to a VLIW compiler: because of space constraints on the microprogram store, trace-style scheduling cannot be applied to every part of a program as a global optimisation, so scheduling must focus on WC path fragments only. The implementation of the first four steps is discussed as follows. Trace formation is examined in section 6.3.1, and operation list building is examined in section 6.3.2. Section 6.3.3 deals with the compaction process, while section 6.3.4 describes the exit cleanup process.

6.3.1 Trace Formation

The purpose of trace formation is the generation of a path through the basic blocks of a program. Conventional superblock schedulers always select the most likely path through untraced basic blocks [48, 87], but the MCGREP-2 implementation does not make any assumptions about

Trace Formation

▷ /tracegen/useprofile.c

Do_Action: Can be used as a Path_Function. Builds a trace path using execution profile information (for ACET reduction).

▷ /tracegen/evaltrace.c

Get_Action: Can be used as a Path_Function. Builds a trace path according to a vector of branch commands.



the nature of the path being traced. Instead, it uses the Path_Function provided by other code, enabling the superblock scheduler to be applied to reduce either ACET or WCET.

Trace formation begins at starting address A . It continues until either a breakpoint address B is reached or a branch is reached at address x . If a branch is reached, then the next address y is computed by calling Path_Function with x supplied as a parameter. The return code selects the next operation:

- **BR_ASSUME_TAKEN:** y is the branch target address. A side exit path is generated for cases when the branch is not used.
- **BR_ASSUME_NOT_TAKEN:** y is the successor of x representing sequential flow, i.e. the branch is not taken. A side exit path is generated for the branch target.
- **BR_ALWAYS_TAKEN:** y is the successor of x representing a taken branch. No side exit path is generated.
- **BR_NEVER_TAKEN:** y is the successor of x representing sequential flow. No side exit path is generated.
- **BR_END_TRACE_NOW:** the path is complete. Tracing stops after the next branch.

Trace formation will also stop if an operation that cannot be part of a trace is reached. In general, this is any instruction word that cannot be decoded into one or more microoperations. Illegal instructions and “branch to microprogram” operations are two examples, but trace formation also stops if a computed jump instruction is reached. (The program counter is set to a register value by computed jump instructions.) A computed jump instruction could lead to any part of the program, and it is not generally possible to work out the destination during trace formation. But correct execution can be preserved by returning to machine code execution immediately before the computed jump instruction is reached, so that it is handled by the machine code interpreter.

Many different implementations of Path_Function can be written for MCGREP-2. Two examples are given below. The first builds a trace according to profile data (for ACET reduction, or WCET reduction in a single-path program). The second builds a trace according to an input vector, allowing external programs to specify the actions to be taken during trace formation.

6.3.2 Operation List Building

The purpose of operation list building is to convert the trace path into a sequence of operations to be carried out during the trace. This sequence is partially ordered by the data dependences of the operations.

In a conventional superblock scheduler, this conversion is carried out using a data flow graph as an intermediate stage (similar to Figure 6.4). Such graphs make it easy to determine the data dependence order, since it is the distance of each node from the root (which represents data sources before the start of the trace).

Within a VLIW code generator, each operation is likely to be encoded in some idealised intermediate form that minimises assumptions about the target platform. For example, it may be assumed that an infinite number of registers are available so that register allocation [2] can be performed during code generation.

For MCGREP-2, the input is machine code, generated using assumptions about the target CPU ISA. This means that register renaming has to be applied to eliminate “false” data dependences that only exist because of register reuse. It also means that information about the program that might be provided by a VLIW compiler in a more conventional arrangement is not available. Some of the information that could be inferred about a program from its source code is destroyed by the RISC code generation process that produced the machine code, such as information about the effective addresses of memory operations.

These factors limit the effectiveness of superblock scheduling, and they force conservative decisions to be made in some instances. For example, it is not safe to change the order of store operations because the same memory location may be updated by two or more store operations. These problems are the same issues faced in an superscalar out-of-order CPU. Such a CPU also extracts as much information from machine code as possible, because it is not possible to rely on an idealised intermediate code.


This suggests that both the operation list and the compaction process should make use of an algorithm for a superscalar out-of-order issue unit. Such algorithms have been discussed in section 2.2.2.2: examples include a scoreboard approach [31], Tomasulo’s algorithm [254], and Sohi’s algorithm [236]. All extract ILP from sequential machine code. Although these algorithms are designed to be implemented in hardware within a superscalar issue unit, and all are designed to operate online using feedback from CPU components (receiving signals such as “operation complete”), they can also be carried out offline in software by using a CPU model to generate feedback. This process emits microoperations and effectively carries out compaction.

To support these algorithms, the operation list that is emitted is sequential machine code in path order, i.e. the contents of each basic block along the trace path. The machine code is decoded into an internal representation by a procedure that makes use of the MCGREP-2 instruction decoder (section 6.2.1): this allows the trace-style scheduler to be used with machine code in any ISA supported by MCGREP-2. The trace path is appended with an “exit” operation causing the program counter to be updated with the address of the next machine code operation. A short path containing only an “exit” operation is generated for each side exit path: this also updates the program counter appropriately.

Operation List Building

▷ /tracegen/isadecode.c

Program_Create: converts machine code into a path, represented as a vector of SSS_Opcode objects, one per microoperation.



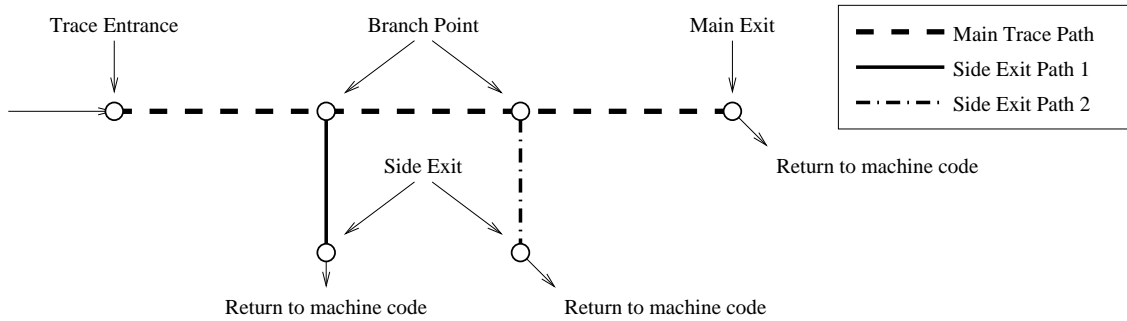


Figure 6.15: Structure of a trace, with two side exits and one main exit. Three microprograms will be generated - one for the main path, and one for each side exit.

6.3.3 Compaction

In general, the purpose of compaction is to generate a microprogram of minimal length to implement the operation list. In MCGREP-2, the task is similar: the aim is to generate a microprogram to implement the operation list on the RFU. Because non-backtracking heuristics are used, the result may be suboptimal [85] in general, but the heuristics have $O(n)$ execution time and give near-optimal results in most cases. During compaction, the operation list is scanned in order, and each operation is packed into the microprogram as early as possible.

In this implementation, compaction generates one microprogram for each exit from a trace. Every trace has one main exit and zero or more side exits, generated as part of the trace path. This structure is shown in Figure 6.15. The operation list for the main path is described in the previous section, and the operation list for each side exit path is initially empty. The side exit paths are populated with operations by the bookkeeping and exit cleanup stage (section 6.3.4).

If a superscalar out-of-order issue algorithm is to be used for microcode generation, one important feature that must be provided by the algorithm is support for precise exception handling (section 6.1.2), sometimes known as “precise interrupt handling” [234]. Precise exceptions always leave the CPU in an execution state that can be resumed without changing program semantics.

While exceptions do not need to be implemented by MCGREP-2 CPUs since they can be avoided in software, precise exception handling is nevertheless needed to support branches within traces. This is because branch mispredictions need to be resolved by a precise mechanism. If a branch is predicted taken, then operations after the branch may be moved before it so that greater ILP can be exposed through speculative execution. But if the branch is not taken (a misprediction) then the speculative executions have to be undone. So the sequential semantics of machine code also need to be preserved in the event of a mispredicted branch.

In principle, every exception is a form of branch misprediction. Each instruction that might gen-

erate an exception could be represented by a rarely-taken branch to an exception handling routine. Because of this similarity, superscalar out-of-order issue algorithms may handle exceptions and branch mispredictions in the same way. Approaches for precise exception handling in out-of-order CPUs are surveyed by Smith and Pleszkun in [234]:

- **In-order completion approach:** instructions are delayed so that they complete in operation list order, although they may begin in any order. Exceptions caused by an instruction are thus suspended until the instruction is ready to complete (i.e. all previous instructions are complete), and each exception causes all uncompleted instructions to be flushed.

This approach creates difficulties when registers or memory are updated by incomplete instructions that are later flushed due to an exception: this could introduce “read after write” hazards. One way to avoid this for the register case is to reserve registers that are used by incomplete operations so that future operations cannot modify them before completion. Store operations can be handled by waiting for all preceding instructions to complete before allowing a store to issue, or by delaying the actual memory write operation until none of the incomplete instructions could cause an exception.

A general disadvantage of in-order completion is that fast instructions must wait for slower instructions to complete. This limits the ILP available. Additionally, reserving registers may sometimes be unnecessary because of false dependences. This also limits ILP.

- **Reorder buffer with bypass logic:** instructions can complete in any order, but the effects of each instruction are committed to the CPU state in operation list order. The reorder buffer is an additional pipeline stage before the register file. Exceptions caused by an instruction are suspended until that instruction is ready to commit, and as before, exceptions cause uncommitted instructions to be flushed.

A reorder buffer approach carries out memory writing operations in order, but there is no need to reserve registers being used by incomplete operations, because register renaming is carried out by the reorder buffer. This allows greater ILP to be exploited by eliminating false dependences. Register renaming is implemented by a bypass network that enables register values to be taken from the reorder buffer instead of the register file. The bypass network is used whenever an operation x following an uncommitted operation y accesses a register that has been updated by y . Unfortunately, this bypass network is one of the disadvantages of the reorder buffer approach, since a large number of comparators are required and much chip space and energy is consumed by this logic. However, this disadvantage does not apply to the MCGREP-2 implementation of precise exceptions, because the effects of the bypass network and temporary registers can be created by static register allocations.

- **History buffer and future file:** both of these approaches are functionally equivalent to the reorder buffer with bypass logic approach, but less logic is required [234]. As no bypass logic will actually be implemented in hardware generated by MCGREP-2, neither approach needs to be considered.

Sohi’s algorithm [236] integrates precise exception handling and superscalar out-of-order issue into a single solution that makes efficient use of hardware. Sohi observed that a single *register update unit* (RUU) could be used for four functions: (1) accepting new instructions from an operation

queue, (2) resolving data dependences as instructions complete, (3) issuing instructions to functional units, and (4) committing the results of an instruction. The fourth function is always applied in operation queue order, so that exceptions can be handled precisely by flushing the RUU. Previous approaches separated the processes of out-of-order issue, data dependence resolution and committing results, and while the effect could be functionally equivalent, implementation would require more hardware.

In MCGREP-2, the hardware usage of the trace compaction algorithm is irrelevant because compaction is implemented in software. But efficiency is important, and one feature of approaches such as a reorder buffer with a bypass network is that many redundant comparisons are carried out. Additionally, it is useful to be able to base the compaction algorithm on a single source, and as Sohi's algorithm has all of the features required for trace compaction in one integrated solution, it is an ideal choice. Furthermore, Sohi's algorithm has been previously implemented in software by the SimpleScalar simulator [41]. The following features set the MCGREP-2 implementation of Sohi's algorithm apart from other implementations:

1. The microoperations provided as input can come from an operation list, either for the main path (produced as described in section 6.3.2) or a side exit path (as described in section 6.3.4). The main path is the default. A side exit path is selected as a result of actions described in point 3 below.
2. On each iteration, Sohi's algorithm produces at most one microoperation to be executed by each of the functional units in the RFU generated by MCGREP-2. These microoperations are stored by the output stage (section 6.3.5) within a microprogram. A more conventional implementation of the algorithm would execute the microoperations immediately.
3. The complete internal state of the implementation is stored within a single context object (`SSS_Context`) which is copied at each point where an exception might occur: two such branch points are illustrated in Figure 6.15. This "fork" operation enables two separate futures to be explored: one in which the exception did not occur, and another in which it did. The "exception" future is characterised by the following operations which are carried out immediately after the fork:
 - Uncommitted microoperations are flushed from the RUU;
 - The output is sent to a different microprogram representing the side exit path;
 - The input is changed to the operation list for the appropriate side exit path, rather than the original operation list.

The "no exception" future is a continuation along the main trace path. No changes are made to the `SSS_Context` object.

4. Dynamic memory disambiguation is omitted. A simple "pseudo queue" scheme [236] is used to ensure that memory operations are carried out in a correct order. Any order is permitted, with the following exceptions:
 - Store operations always happen in the original order.
 - Store operations cannot issue until all branches have been committed.
 - Load operations cannot be reordered across store operations.

Compaction

```

  > /tracegen/sssopt.c
  SSS_Optimiser: implementation of Sohi's algorithm applied to
  the trace compaction problem.

```



In other implementations of Sohi's algorithm, some of these rules can be relaxed by checking the effective addresses of load and store operations. However, effective addresses are not known during trace building, so conservative choices have to be made in order to preserve functional correctness. This causes a reduction in available ILP, but this appears to be inevitable due to the requirement for operation that is statically predictable.

During compaction, data is assigned to physical registers. This is done for constant values, such as the immediate values attached to some machine code instructions, and for temporary variables being used by the RFU. Constant registers can be replicated across all units of the RFU, since their values are never updated, but variable registers cannot easily be replicated in this way because of the limited number of ports on each register file.

Handling variable registers is therefore a form of *register clustering* problem [87], because each variable register can only be present in a subset of all register banks. The clustering scheme that is most easily implemented places copies of each variable register in the two register banks associated with the functional unit that computed it.

In some clustered VLIW architectures, each functional unit only has direct access to a local register file. Access to other register files is possible via an interconnect, but there is an additional time cost associated with this, and an interconnection resource is consumed. On such architectures, instruction schedulers attempt to allocate operations to functional units in order to minimise these costs. This is a hard problem because the number of possible allocations grows exponentially with linear increases in program size, so heuristics are used. The *bottom-up greedy* (BUG) cluster assignment algorithm described by Ellis [74] is an early example. BUG scans the operation list in reverse order ("bottom up") before compaction begins, and adds hints to each operation to indicate the most suitable functional unit for that operation. BUG assumes that register file ports are the only limiting resources: other limits such as interconnection bandwidth and register space are disregarded. The more advanced *partial component clustering* (PCC) [81] approach partitions an operation list into groups of operations to be scheduled on the same functional unit: the members of each group are chosen to reduce the inter-group communications required.

However, each functional unit generated by MCGREP-2 can access any register file port at the same cost, and interconnection links are not shared. Therefore, register file ports are the only bottleneck affecting register access. Register cluster assignment algorithms do not need to be used because every functional unit has equivalent access to each register file port. The use of a more restrictive interconnect would force the introduction of an assignment algorithm such as BUG or PCC, but this is not required for MCGREP-2 at present.

Exit Cleanup

```
▷ /tracegen/program.py
```

Terminate_Queue: generates an exit cleanup routine to end a path. (Python prototype.)

```
▷ /tracegen/isadecode.c
```

Terminate_Queue: generates an exit cleanup routine to end a path. (C implementation.)

**6.3.4 Bookkeeping and Exit Cleanup**

Because superblocks are used, side entrances do not need to be considered, simplifying implementation [48, 87]. But exit paths do need some special handling to ensure that the function of the trace matches the function of the machine code that it replaces. To this end, the operation list for each path leading to an exit is extended with the following:

- **Register writeback:** values are copied from temporary register locations in the RFU into the RISC register locations.
- **Flag writeback:** the CPU condition flag is updated with the result of the last comparison carried out during the trace, if that comparison flag has not been consumed by a branch operation. ORBIS32 has a single condition bit. Other ISAs may have several or none. For example, the 68000 CPU has five condition bits (carry, zero, overflow, extend, and negative [181]). All five are set by ALU operations, and setting them correctly is essential to support subsequent branch operations.
- **Repeat:** if the trace exit address is the same as the trace entrance address, then a micro-branch operation is added to make the trace repeat itself without returning to machine code. This speeds up the operation of looped code by skipping two instruction fetches. If a repeat operation can be added, no more operations will follow it.
- **Program counter update:** modifies the program counter by adding or subtracting an offset so that machine code execution is resumed at the correct address.
- **Return to machine code:** causes the CPU to return to machine code operation.

In the compaction stage, these operations are abstract representations, acting as markers for the output stage.

6.3.5 Output Stage

In accordance with common software engineering practice, the details of the MCGREP-2 architecture are hidden from the trace generating stages as far as possible. The purpose of the output stage is to act as an interface between the trace generator and the MCGREP-2 microprogramming interface (described in section 6.2.4). The abstraction is independent of the version of the CPU being used: for example, the same functions are used regardless of the size of the RFU array. The abstraction

Trace generator back ends

▷ `/tracegen/mcgrepinf.c`

Backend that generates MCGREP-2 microprograms, sending the output to a file (for use on a PC) or directly to the microprogram store (for use on CPUs generated by MCGREP-2).

▷ `/tracegen/harnessinf.c`

Backend that is used by the `harness.py` trace generator test program (section 6.4.2).

▷ `/tracegen/microinf.h`

Both backends implement this interface.



also allows the microprogram output stage to be replaced by a test output stage that verifies the correctness of the test. The key types of function provided by the output stage are:

- Architecture query functions, such as `Microprogram_Get_Num_Units` and `Microprogram_Unit_Is_Suitable`. These enable the trace generator to obtain necessary information about the RFU without calling MCGREP-2 interface functions directly.
- Operation programming functions, such as `Microprogram_Port`. These modify the microprogram to set up a particular operation.
- Control flow functions, such as `Microprogram_Repeat_Cycle`. These generate microprogram branch instructions.
- Register management functions, such as `Microprogram_Allocate_Immediate` and `Microprogram_Allocate_Reg`, which allocate and free space with the register files. Registers within the RFU are used to store constants, many of which represent the immediate values that were attached to machine code instructions. This register space has to be updated at the same time as the microprogram store when new microcode is being loaded.
- Microprogram management functions, such as `Microprogram_Upload` which uploads a microprogram to the RFU microprogram store (this routine only works when executed within the simulator or hardware).

The output stage hides the details of microprogramming from the rest of the trace scheduling implementation. The trace scheduler is not aware of the operations required to begin and end the execution of a custom microprogram, such as flushing the LSU at the start of execution, and branch to the system cycle on completion.

In turn, the actual bit manipulations needed to program operations and control flow are hidden from the output stage by the MCGREP-2 microprogramming interface. This is essential because the purpose of each microcode bit is dependent on the MCGREP-2 configuration: a fact which is hidden from the trace-style scheduler by the interface layer.

6.4 MCGREP-2 Evaluation

This section evaluates MCGREP-2 against the requirements for the work and the hypothesis (section 3.4). The components implemented within this chapter are shown as part of the abstract WCET reduction process in Figure 6.16. Implementations of TARGET generated by MCGREP-2 are evaluated in three ways:

- Against the TARGET parameters: what subset of TARGET does MCGREP-2 implement? (section 6.4.5);
- Against the requirements: how well does MCGREP-2 meet the requirements for the work? (sections 6.4.6 to 6.4.12); and
- For correctness. This part of the evaluation is expanded to incorporate testing of machine code operation (section 6.4.1), the microcode generator (section 6.4.2), machine code and RFU operation together (section 6.4.3), and RFU operation in hardware (section 6.4.4).

Notes on possible improvements are given in section 6.4.13, and the evaluation is summarised in section 6.4.14. As before, the programs and hardware designs used to carry out these evaluations are part of Appendix A. Build instructions can be found in sections A.7 through A.10.

As in section 5.3, benchmark programs are used for many of the experiments in this section. The MCGREP-1 benchmark programs are reused, since the evaluation criteria are unchanged for MCGREP-2, but there are two changes to the benchmark set.

Firstly, the `coverage_test` benchmark was removed because the conditions that it tested are no longer supported on CPUs generated by MCGREP-2, specifically alignment exceptions and interrupts. If support for these features is added to MCGREP-2, then new tests will be required. Extensions to MCGREP-2 for hardware floating point support would also allow more MIBench/Mediabench program to be used.

Secondly, it is possible to add some more benchmarks from the MIBench/Mediabench corpus [108, 154]. One of the problems with the MCGREP-1 experiments was caused by memory limitations, which prevented the use of certain benchmark programs. However, the MCGREP-2 simulator can permit limited access to the host filesystem through a “system call” interface. This allows benchmarks to read and write files, increasing the number of benchmarks that can be executed in the simulator. The programs that make use of this feature are marked with a (*) in Table 6.6: these cannot be executed on hardware generated by MCGREP-2 as the system call feature is not supported.

This change means that more types of benchmark can be used to test MCGREP-2. As in section 5.3, the primary focus is on ensuring that MCGREP-2 operates correctly, and the additional benchmarks improve the quality of the testing.

6.4.1 Correctness Evaluation: Machine Code Operation

Machine code execution on CPUs generated by MCGREP-2 is tested in two ways: (1) by comparing the MCGREP-2 simulator against the reference OpenRISC simulator, and (2) by comparing the MCGREP-2 simulator against the hardware. Both types of test make use of benchmark programs listed in Table 6.6.

The first test operation is carried out as illustrated in Figure 5.14(a): a test harness executes both simulators together and detects differences in machine state. The second test operation benefits

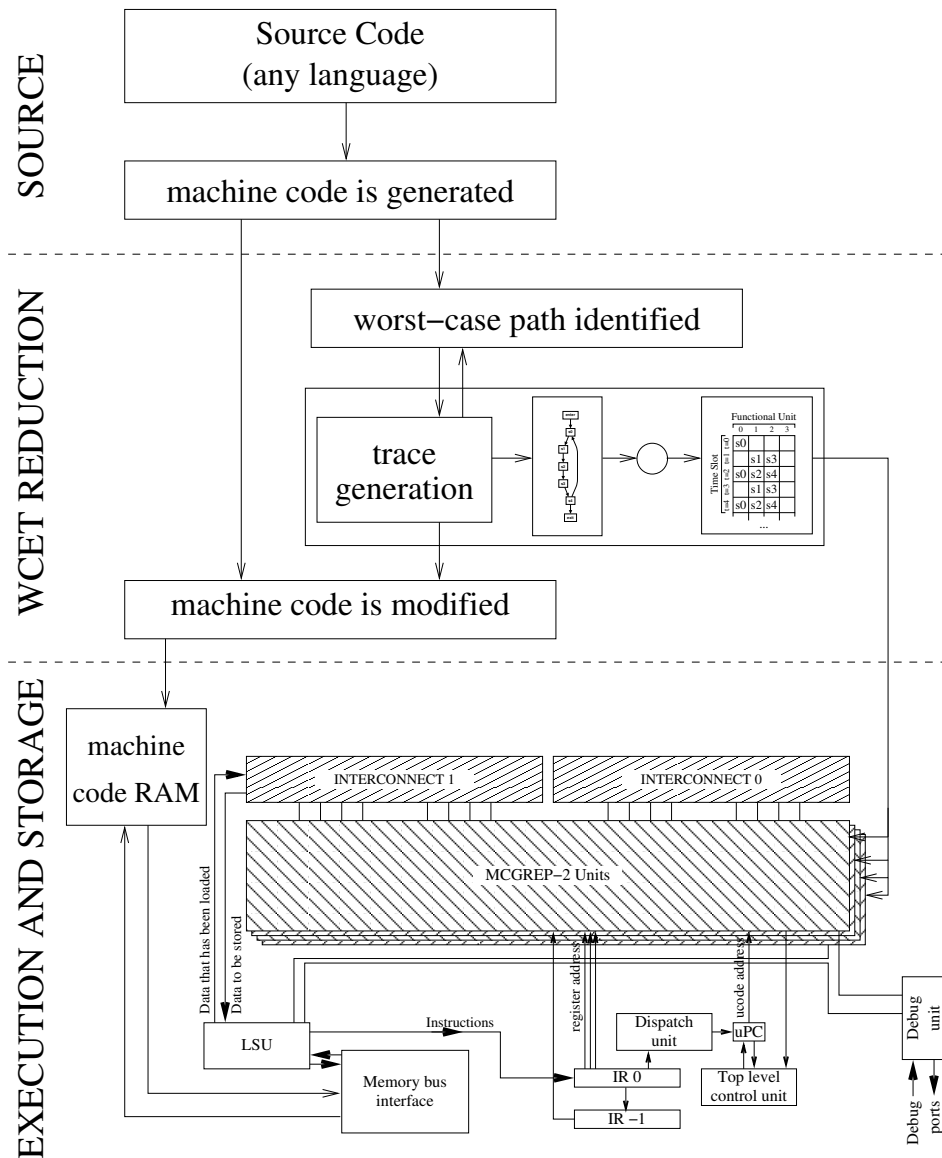


Figure 6.16: The MCGREP-2 components shown as parts of the abstract WCET reduction process. The automatic microcode generator is shown as a part of the WCET reduction level. The MCGREP-2 hardware is shown within the storage and execution levels.

Benchmark	Purpose	Note
aes	Encryption algorithm	
bitcount	Counts bits set to 1 within an array	
blowfish	Encryption algorithm	(*)
crc32	Checksum computation	
dijkstra	Shortest-path algorithm	
g721	Telephone-quality voice compression (1)	
gsm	Telephone-quality voice compression (2)	(*)
jpeg	Image decompressor	
mad	MPEG audio decompressor	(*)
patricia	Network routing algorithm	(*)
qsort	Sort algorithm	
sha	Cryptographic hash algorithm	
stringsearch	Finds substrings with strings	(*)

Table 6.6: Benchmarks used as a test corpus for MCGREP-2 CPUs. Note (*): These benchmarks use a SimpleScalar-like “system call” interface to access files on disk, and are therefore not suitable for testing in hardware, or comparison with the OpenRISC simulator.

from the debugging harness features of the MCGREP-2 hardware (section 6.2.2.6). The MCGREP-1 tests (section 5.3.1) required manual comparison of the hardware and software through the debugging monitor interface (Figure 5.16). For MCGREP-2, the same comparisons are performed automatically by the `hardware_test` program, which detects differences between the state of the MCGREP-2 simulator and the state of the hardware as both execute the same program.

Each comparison is carried out by downloading the current state of the debug chain, then decoding it into a set of register values by reusing the same MCGREP-2 model used to generate the VHDL. After each successful comparison, the simulator and hardware are both advanced by N clock cycles. The comparison is then repeated. The testing tool permits the value of N to be changed during execution, so it is possible to perform the tests at a coarse grained level (with large N) or at a fine grained level (with $N = 1$). The debug chain is used to check the ALU outputs, instruction word, and uPC value. However, fine grained testing is very slow due to the overhead of downloading the debug chain state after every clock cycle. Therefore, tests of MCGREP-2 hardware were only performed at the fine grained level for debugging purposes.

The hardware comparisons were performed by building the MCGREP-2 hardware for the Avnet/Memec MM1 “mini module” FPGA board [22] (Figure A.2), which provides a Spartan-3 FPGA. This board is used in place of the Spartan-2E “BurchEd B5” FPGA board used to test MCGREP-1 because a larger amount of on-board RAM is available (1 Mb instead of 256 Kb) and the FPGA device has more features (such as larger block RAMs and multiplier cores). Both types of test were successful, indicating that execution of machine code on MCGREP-2 CPUs is the same as OpenRISC for the benchmarks tested.

6.4.2 Correctness Evaluation: Microcode Generator

The trace-style microcode generator described in section 6.3 can be validated in two ways: (1) by execution, and (2) by abstract interpretation (AI). In this section, the second approach is used, since it is independent of the data being processed by the operations. Therefore, a false positive

Machine Code Tests

```
▷ /testcases/orlkcompare.sh
```

Test script: builds benchmark programs, then compares the execution of each within the MCGREP-2 simulator against execution within the OpenRISC simulator.

```
▷ /testcases/hwcompare_openrisc.sh
```

Test script: builds benchmark programs, then compares the execution of each within the MCGREP-2 simulator against execution within MCGREP-2 hardware on FPGA.

```
▷ /mcg2-src/vhdl/build.ref.hw
```

Builds FPGA programming file for MCGREP-2 CPU on MM1 FPGA board [22].




result cannot be generated because of a particular combination of data values, causing a test based on comparisons of register values or memory contents to pass when the underlying operations are incorrect. This data independence also makes AI useful for WCET calculations (section 2.1.2), but in this case, AI is applied on a much smaller scale within a single trace. The following procedure is repeated many times:

- Select a machine code fragment at random from a test corpus.
- Select random parameters:
 - Number of functional units,
 - Trace length (minimum number of instructions),
 - Memory access latency,
 - Path through machine code (this is selected by using a random “branch predictor” at each branch point, which results in each branch being taken with 50% probability).
- Generate microcode using the parameters.
- For both the microcode and the input machine code:
 - Let the abstract value of register x be represented by $r(x)$.
 - Initialise all RISC register values. For all $0 \leq x < 32$, $r(x) = x$.
 - When an operation A is applied to registers x and y , with the result stored in z , then $r(z)$ is redefined as follows: $r(z) = (A, r(x), r(y))$.
 - Register writeback operations are considered to be direct copies (i.e. $r(z) = r(x)$).
- The final value of every $r(x)$ for $0 \leq x < 32$ must be the same for both trace and machine code interpretation. Testing for equality guarantees that sequential execution and traced execution are equivalent along the chosen path, regardless of data input, because the resulting tuples in each $r(x)$ indicate the path taken by abstract data through the trace.

Microcode Generator Tests

```

> /tracegen/harness.py
Microcode testing tool using AI.
> /tracegen/frags
Test corpus: machine code fragments from benchmark software.
```



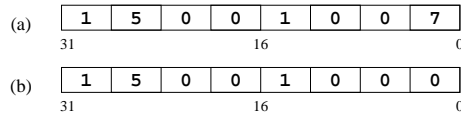


Figure 6.17: (a) The checkpoint instruction has no effect on the CPU state or RAM: it is interpreted as a NOP. However, it causes the simulator to execute a checkpoint procedure to capture the CPU state and RAM contents. (b) The hotspot marker instruction can also be interpreted as a NOP, but it is used to identify starting points for custom microcode.

This test successfully demonstrated that the trace-style microcode generator's operation was correct (i.e. the RFU microcode is functionally equivalent to sequential machine code). However, the test is incomplete without also considering the correctness of the MCGREP-2 microprogramming interface and integration of RFU features into programs. These tests are carried out by execution as described in the next section.

6.4.3 Correctness Evaluation: Machine Code and Microcode

For the tests in the following section, each test program in Table 6.6 was augmented by one or more custom microprograms. The machine code fragments implemented by these microprograms were selected using profiling data. In each case, the number of branches represented in the microcode was selected so that less than 512 microinstructions would be required with an array size of 3: this facilitates comparisons using hardware generated by MCGREP-2, where the microprogram store size is limited.

Previously, the correctness of microprograms has been checked by comparing the effects of executing machine code with the effects of executing microprograms within a simulator. In MCGREP-1, this was possible because of the clear boundary between microprograms and the surrounding code, since microprograms always replaced machine code within a continuous address range. Each comparison could be carried out when both the microprogram and the original machine code crossed the boundary defined by that range. But the continuous address range restriction does not apply to microcode generated by a trace-style scheduler. So, while it is easy to detect an exit from a trace by watching for the instruction decoder's operation, it is not easy to identify the corresponding point in pure machine code execution.

To avoid this problem, the notion of checkpoints was introduced. A checkpoint is a machine instruction, introduced by adding the macro `CHECKPOINT` to C source code. The machine code is shown in Figure 6.17(a). When the program is executed within an extended version of the MCGREP-2 simulator, this code causes the state of the RAM to be captured and stored using the

MD5 cryptographic hash function [225]. The state of the CPU registers are also stored: hashing is not applied to these so that their exact values can be reported in the event of test failure. The use of a MD5 hash function allows single-bit differences in registers or memory to be detected reliably without needing to store complete copies of the RAM state. Only 128 bits need to be stored at each checkpoint, plus $32 \times 32 = 1024$ bits for the register state.

Checkpoint tests require a program to be executed twice. On the first run, custom microcode is not used, and checkpoint values are stored in a list. On the second run, custom microcode is used, and checkpoint values are compared to the list after being computed. A mismatch causes the test to fail. Checkpoints detect register update errors, memory errors, and incorrect paths being taken through the program. They can be added to any point in a program, but cannot be part of a trace due to the effects of speculative execution. Tracing stops if a checkpoint is reached, so it is best to place checkpoints away from code structures that are likely to be optimised as WC path fragments because they reduce the effectiveness of tracing.

A related strategy is used to specify the start points of custom microcode. MCGREP-1's solution to this problem was to extract machine code fragments that contributed significantly to the length of the WC path, then optimise them separately, and finally patch the program machine code with a "branch to microprogram" instruction (Figure 5.10). This approach has the advantage of allowing any sequence of instructions to be partly replaced by a microprogram, even if the source code is not available (e.g. C library functions, where source exists but recompilation is not convenient). But this approach is hard to use, because changes to the source code of a function along the WC path will force the process to be repeated. Additionally, some machine code includes absolute addresses of objects in memory, so just recompiling the program with new code elsewhere may also cause problems.

For MCGREP-2, a more user-friendly scheme is introduced. Programmers can place the `HOTSPOTMARKER` macro at any point within code. This introduces a special instruction (Figure 6.17(b)) which is interpreted as a type of NOP. Occurrences of `HOTSPOTMARKER` are identified by a post-compilation search pass which uses profile or WC path information to build traces beginning at each marked location, following the most likely or worst-case path. Once the microprogram associated with each trace has been uploaded into the microprogram store, each `HOTSPOTMARKER` is replaced with an appropriate branch to microprogram instruction. A sample usage of `HOTSPOTMARKER` is shown in Figure 6.18.

Because both `HOTSPOTMARKER` and `CHECKPOINT` introduce additional instructions (NOPs), they can increase the execution time of programs. This is true in every case where `CHECKPOINT` is used, since it cannot be part of a trace, and it is also true when `HOTSPOTMARKER` is used but microcode is not generated. Therefore, both `CHECKPOINT` and `HOTSPOTMARKER` can be completely removed from a program by using the `#define` command. This is done in order to measure machine code execution times in the absence of custom microcode for the comparisons in section 6.4.8.

Unfortunately, `HOTSPOTMARKER` alone is not enough to reduce the ACET or WCET of some programs because the hotspots are actually located in C library functions. For example, most of the execution time of `qsort` is spent in the comparison function `strcmp`. For this reason, the MCGREP-2 tools also support MCGREP-1 machine code patches, which are used in addition to any explicitly marked hotspots.

All of the test programs in Table 6.6 were extended with `HOTSPOTMARKER` and `CHECKPOINT` markers at appropriate places in their source code. Then, the following experiment was carried out:

```

while (qcount( queue ) > 0)
{
    dequeue (&iNode, &iDist, &iPrev);
    for (i = 0; i < NUM.NODES; i++)
    {
        HOTSPOTMARKER ;
        if ((iCost = AdjMatrix[iNode][i]) != NONE)
        {
            if ((NONE == rgnNodes[i].iDist) ||
                (rgnNodes[i].iDist > (iCost + iDist)))
            {
                rgnNodes[i].iDist = iDist + iCost;
                rgnNodes[i].iPrev = iNode;
                enqueue (i, iDist + iCost, iNode);
            }
        }
    }
}

```

Figure 6.18: A `HOTSPOTMARKER`, added to the inner loop of the `dijkstra` benchmark program [108]. The most likely execution path through the two `if` statements will be optimised using a custom microprogram.

1. **Experiment Goal:** To demonstrate that machine code and custom microcode are functionally equivalent.
2. **Software Setup:** The MCGREP-2 simulator is extended to support the `CHECKPOINT` instruction (Figure 6.17(b)).
3. **Method:** For each program in Table 6.6, the following process was carried out:
 - (a) Execute the program, but replace all `HOTSPOTMARKER` instructions with NOPS so that custom microcode is not executed. During this execution, record the system state at every checkpoint. (This is achieved using a “NOP hook” - see Table 6.5.)
 - (b) Execute the program, without replacing any `HOTSPOTMARKER` instructions. During this execution, check the system state against the previously recorded state. A mismatch indicates that the test has failed.
 - (c) Repeat the experiment for different array sizes (numbers of functional units).

The experiment is successful if every check is an exact match.

4. **Results:** This checkpoint experiment was successful (Table 6.7). It was performed for array sizes from 1 to 4 units. The use of checkpoints indicates that traces are functionally indistinguishable from machine code, at least in the cases of the benchmark programs tested.
5. **Evaluation:** This experiment provides an easy way to ensure that traces and machine code are functionally equivalent for a specific subset of programs.
6. **Conclusion:** The results indicate that MCGREP-2 traces works correctly as part of a larger program.

Program	Checkpoints Tested
aes	2002
bitcount	9
blowfish	54571
crc32	22
dijkstra	102
g721	147521
gsm	134
jpeg	802
mad	155
patricia	10891
qsort	85833
sha	81526
stringsearch	58

Table 6.7: Trace and machine code execution were compared for each of the programs listed in this table. Each checkpoint test was an exact match.

Microcode vs. Machine Code

▷ /testcases/hskit.py

Test framework: for each array size and program, identifies marked hotspots, calls the trace generator to build traces, then executes each program with and without traces for checkpoint testing.



RFU Hardware Tests

```
▷ /testcases/hwcompare.openrisc.trace.sh
```

Test script: builds benchmark programs with custom microcode, then compares the execution of each within the MCGREP-2 simulator against execution within MCGREP-2 hardware on FPGA.

```
▷ /mcgprep2-src/vhdl/build.ref.hw
```

Builds FPGA programming file for MM1 FPGA board [22].



6.4.4 Correctness Evaluation: Microcode in Hardware

Machine code execution on the MCGREP-2 hardware has already been tested (section 6.4.1), but it is also important to ensure that custom microprograms operate correctly. To do this, a 3 unit MCGREP-2 CPU was built for the Spartan-3 FPGA on the MM1 FPGA prototyping board [22]. Then, the `hardware_test` program was used to compare the operation of the simulator and the operation of the hardware, as benchmarks from Table 6.6 were executed. Each test was successful, with no errors detected, so the MCGREP-2 hardware acts the same as the MCGREP-2 simulator. It is therefore possible to use the simulator in place of real hardware during experiments, as measurements produced using real hardware are indistinguishable from simulated results.

6.4.5 Evaluation against the TARGET parameters

In this section, the capabilities of CPUs generated by MCGREP-2 are compared to TARGET in order to show how well MCGREP-2 represents all possible instances of TARGET. Many of these comparisons make references to an extension to MCGREP-2 and the trace generator to support the ISA of the Microblaze CPU [286]. This extension is described in Appendix D. It demonstrates that the parameterised features of MCGREP-2 can be adjusted to the extent of supporting an entirely different ISA.

The planned parameterisable features for MCGREP-2 are listed in Table 6.2. Each feature is evaluated as follows:

- **ALU function set:**

ORBIS32 and Microblaze include many similar ALU functions, but differences do exist. For example, the Microblaze subtract operation (`rsub`) reverses the parameters of the computation versus ORBIS32 (i.e. $B - A$ rather than $A - B$). Microblaze also does not specify a large number of explicit comparison operations: rather, comparison operations are carried out at the same time as branching. These differences mean that an entirely different set of ALU functions are effectively needed for each ISA. However, as the Microblaze implementation indicates, support for these features can be built into MCGREP-2 by creating an alternate architecture definition.

There is no apparent limit to the range of ALU functions that could be accommodated by MCGREP-2, which acts as a sort of container, providing an infrastructure to set ALU control lines according to the specification encoded in a microprogram. The actual function carried

out in response to the microprogram is fully extensible using VHDL (for hardware) and C (for the simulator), and it is conceivable that extremely complex ALU functions could be implemented as ASIP-style ISA extensions.

- **ISA, Built-in Microprogram and Microprogram Interface:**

Supporting an alternate ISA also requires a new instruction decoder and a new built-in microprogram. The architecture definition allows both of these to be fully redefined, making use of common components such as the instruction decoder generator and the microprogram interface which are shared between both the Microblaze and ORBIS32 architectures.

The ISA decoder does include an inherent assumption that instructions have a fixed width. Therefore, the capabilities of MCGREP-2 to implement instances of TARGET are limited to RISC-like ISAs with a fixed instruction size. In general, any ISA making use of multiple addressing modes, including x86 and 68000, would require substantial changes to the MCGREP-2 software. A more advanced type of microprogramming interface would be required to support these, probably making use of a two-level microprogram scheme as used within the 68000 [257] to save space.

- **Microprogram Store Dimensions and Register File Dimensions:**

As in MCGREP-1, both of these parameters can be adjusted by extending the driver that produces the block RAM implementation. The driver is selected within the architectural configuration file, which currently provides options for Spartan-2E (RAMB4) and later (RAMB16) FPGAs. Additional drivers could allow larger memories to be selected.

- **l, m, n values:**

MCGREP-2 permits the total $l + m + n$ to be adjusted using the `num_units` parameter in the architectural configuration file. (Figure 4.5 shows how these parameters relate to the RFU dimensions.) While l is unrestricted, the values of m and n are fixed as follows:

- $n = 1$. As section 6.2.2.4 notes, there is no point in introducing additional units that are capable of acting as CPUs, because multiple thread execution will be limited by the memory bottleneck.
- $m = 0$. In CPUs generated by MCGREP-2, only the first functional unit in the array has access to the memory bus. There is no point in allowing access from additional units unless the memory architecture is also improved, since multiple memory accesses cannot be performed concurrently.

MCGREP-2 implements a much larger subset of TARGET than MCGREP-1, but some architectural parameters are still fixed. These are characterised as features implemented within the non-extensible core of MCGREP-2: the part that is not dependent on the target hardware or the ISA selection. One of these is the interconnection between functional units: this is fixed as described in section 6.2.2.

An improved version of MCGREP-2 could be developed to address this, but because CPUs generated by MCGREP-2 already have support for predictable execution of machine code and custom microcode, it is not clear whether this would actually bring any benefits to the intended application of reducing the WCET of hard real-time software. Although MCGREP-2 CPUs are not perfect implementations of TARGET, they are close enough to the ideal to be usable as a demonstration of scalable WCET reduction capabilities.

Group	Example	Clock cycles
ALU (Immediate)	<code>l.addi</code>	$1 + \max(1, M)$
ALU/Shift (Reg)	<code>l.add</code>	$1 + \max(1, M)$
Compare	<code>l.sfeqi</code>	$1 + \max(1, M)$
Jump	<code>l.bnf</code>	$1 + \max(1, M)$
Load	<code>l.lwz</code>	$2 + 2 \times \max(1, M)$
Move High	<code>l.movhi</code>	$1 + \max(1, M)$
NOP	<code>l.nop</code>	$1 + \max(1, M)$
Procedure Call	<code>l.jal</code>	$1 + \max(2, M)$
Shift (Immediate)	<code>l.slli</code>	$1 + \max(1, M)$
Store	<code>l.sw</code>	$2 + 2 \times \max(1, M)$

Table 6.8: Instruction timings on MCGREP-2 CPUs using ORBIS32 ISA. M is the memory latency in clock cycles.

6.4.6 Evaluation against the Requirements

Like MCGREP-1, MCGREP-2 is not representative of all forms of TARGET, but it does provide information about how well TARGET meets the requirements from section 3.2. The effectiveness of CPUs generated by MCGREP-2 as implementations of TARGET is examined in the following sections by reference to the requirements:

- Basic block timing invariance (section 6.4.7);
- WCET reduction versus previous work, and support for general software programs (section 6.4.8);
- Efficient optimisation process (section 6.4.9);
- Scalability, so that WCET reductions can be applied to programs of any size (section 6.4.10) and greater ILP can be used by larger array sizes (section 6.4.11).

6.4.7 Invariant basic block execution times

As for MCGREP-1, basic block timing invariance can be demonstrated by showing that the time taken for machine code and microcode execution is independent of the previous state. Table 6.8 shows the time taken to execute each instruction on CPUs generated by MCGREP-2. (The equivalent for MCGREP-1 is Table 5.10.) This table was obtained by using the MCGREP-2 simulator to execute each instruction within a test harness.

The timing rules in Table 6.8 enable the execution time of a basic block to be calculated by simple addition. The calculation depends only upon the value of M , the memory latency. This property can be demonstrated by a simple experiment:

1. **Experiment Goal:** To demonstrate that the execution time of a basic block can be calculated by adding the execution times of its constituent instructions.
2. **Software Setup:** Since the MCGREP-2 simulator has been shown to be equivalent to the hardware (sections 6.4.1 and 6.4.4), the simulator is used for this test.

Basic Block Timing Invariance Experiment



▷ /testcases/bb_inv_test.py
Experiment program.

3. **Method:** The following process was repeated 100 times, with memory latency values in the range $0 \leq M \leq 9$. A random basic block was constructed by:

- (a) Randomly selecting 10 non-jump instructions listed in Table 6.8,
- (b) Appending a random jump instruction,
- (c) Appending a non-jump instruction for the delay slot.

The execution time cost of the basic block was evaluated by:

- (a) Applying the rules listed in Table 6.8 to obtain the *computed cost*,
- (b) Executing the basic block using the MCGREP-2 simulator to obtain the *measured cost*.

The experiment is successful if every computed cost is equal to the associated measured cost.

4. **Results:** The experiment was successful: for each random basic block, the computed cost matched the measured cost.

5. **Conclusion:** The results indicate that basic block execution times can be calculated using the rules in Table 6.8. Since these rules are independent of all external factors except M , the memory latency, this also demonstrates basic block timing invariance.

Each custom microprogram generated by the software described in section 6.3 may implement more than one basic block, so it might seem that the use of traces acts against the requirement for basic block timing invariance. However, the custom microcode is also composed of basic blocks at the microprogram level, and each one of these basic blocks has a constant execution time. Custom microcode execution is independent of the previous state of the CPU, and microprograms produced by the microcode generator do not wait for CPU components to become ready: execution transitions from one microprogram state to the next independently of other events. Therefore, custom microcode also preserves basic block timing invariance.

6.4.8 WCET Reduction versus Previous Work

In section 5.3.6, microprogrammed execution on MCGREP-1 is evaluated against machine code execution only, on Microblaze and OpenRISC. That section makes two assumptions: (1) that each benchmark is a single-path program, therefore ACET is equivalent to WCET, and (2) that comparison is possible with a CPU that uses a locked cache or instruction scratchpad.

The same assumptions are reused here, even though practical embedded systems software is unlikely to be single-path unless a specialist compilation strategy has been used [204]. The rationale is that the results can be applied to more complex software using WCET analysis approaches due to the basic block timing invariance property of MCGREP-2 CPUs. As in section 5.3.6, it is not

assumed that the benchmarks are representative of all possible embedded systems code: rather, they are used only as examples to ensure that WCET reductions are possible using MCGREP-2.

Earlier work such as Table 5.13 indicates that CPUs generated by MCGREP-1 are comparable with other soft cores such as OpenRISC and Microblaze. As MCGREP-2 CPUs are at least as fast as CPUs generated by MCGREP-1 (contrasting Tables 5.10 and 6.8), this feature is also applicable to MCGREP-2 CPUs. Therefore, experiments only need to compare custom microcode execution with machine code execution on an MCGREP-2 CPU. This will be enough to demonstrate execution time reductions.

1. **Experiment Goal:** To demonstrate that custom microcode can reduce the execution times of programs on an MCGREP-2 CPU, assuming that large scratchpads are used in place of external RAM.

There are two variables. The first is the benchmark program: this is taken from Table 6.6. The second is whether custom microcode is used or not. Two other variables (the array size and the microprogram store size) are assigned fixed values of 3 units and 512 microinstructions respectively.

2. **Software Setup:** The MCGREP-2 simulator is extended with a new instruction, which is used to indicate that microprogram store initialisation is complete.

A 3 unit CPU is simulated by MCGREP-2 for the ORBIS32 ISA. The memory latency M is set to 1, reusing the assumption of section 5.3.6 that the CPU is connected to very large instruction and data scratchpads.

3. **Method:** For this experiment, the execution time of each benchmark program was measured using the MCGREP-2 simulator. Measurement began immediately after the microprogram store was initialised with the relevant traces, and ended as the program completed.
4. **Results:** Figure 6.19 shows the execution times of each benchmark on each platform. The values are normalised to the execution time of the benchmark on an MCGREP-2 CPU in “machine code only” mode.
5. **Evaluation:** The results indicate that custom microprograms do reduce execution times, but the effectiveness of the approach is dependent on the program. The benefit seen for some programs such as `dijkstra`, `crc32` and `aes` is much greater than the benefit for others such as `qsort`. Superblock-style traces include the implicit assumption that control flow is biased within hotspots, which is true for the programs where the approach was most effective, but false for the others. Treeregion formation [87] is one solution for this problem which is not currently implemented in the trace-style microcode generator: in a treeregion, superblocks can be expanded in more than one direction at each branch. Hyperblocks [170] are another solution, since they replace some control flow with predication, which is ideal for handling small numbers of conditional instructions within a trace.

However, since execution time reduction has been demonstrated in every case, the software is suitable for WCET reduction experiments “as is”.

6. **Conclusion:** Custom microcode can be used to reduce the execution time of all of the benchmark programs listed in Table 6.6.

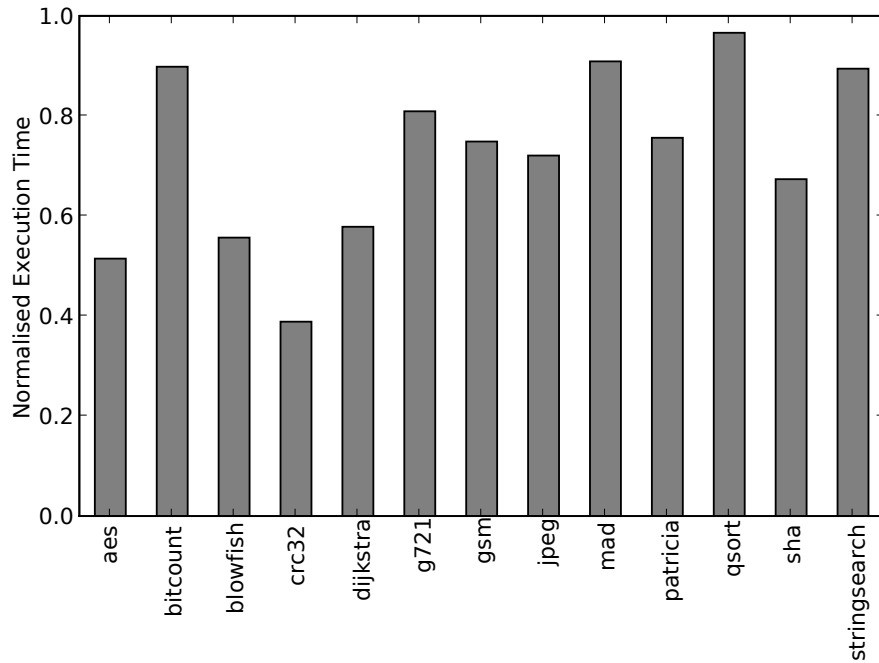


Figure 6.19: Normalised benchmark execution times on MCGREP-2 CPUs with traces. The MCGREP-2 configuration used here has 3 units and space for 512 microinstructions, and each result is normalised to the execution time of the same benchmark using machine code only. (Raw data appears in Table B.3.)

Execution Time Experiment

▷ /testcases/hskit.py

Experimental measurement tool.



Opcode	Purpose
l.nop 0x1011	Marks the beginning of the operation list building process (section 6.3.2).
l.nop 0x1012	Marks the end of operation list building.
l.nop 0x1001	Marks the beginning of the trace compaction process (section 6.3.3).
l.nop 0x1002	Marks the end of compaction.

Table 6.9: New measurement opcodes for timing the operation of the two steps of the trace generation process. Each triggers a measurement function via a simulator hook (Table 6.5).

6.4.9 Efficient optimisation process

Efficient generation of custom microprograms is important because the microcode generator may need to be used as part of a larger search process for WCET reductions. MCGREP-1 did not meet this requirement because manual intervention was required for microcode generation.

MCGREP-2 makes use of an algorithm for microcode generation that is known to be efficient. In principle, trace scheduling is an $O(n)$ operation for a path of n instructions. But this does not provide any information about how fast the generation process actually is, only how well it scales.

Microprograms will normally be produced on a workstation along with other parts of an embedded real-time system. However, all parts of the microcode generating software can be executed on MCGREP-2 itself, and this provides a useful way to measure the time taken to produce microcode as an exact number of clock cycles. Each result is more easily reproduced than a measurement on a workstation, and comparison with other software for MCGREP-2 CPUs is possible (e.g. benchmark programs).

1. **Experiment Goals:** To measure the execution times of MCGREP-2 microcode generating software in various conditions. Two variables are considered here:
 - The benchmark program (from Table 6.6),
 - The MCGREP-2 configuration (1 to 4 units).

The number and length of traces varies from program to program, so the results must specify this information.


2. **Software Setup:** The MCGREP-2 simulator is extended with the new instructions shown in Table 6.9, allowing the time taken for the two steps of each trace generation operation to be measured.
3. **Method:** Each benchmark was executed, with MCGREP-2 array sizes from 1 to 4 units, and each step was timed. The memory latency was fixed at 1 clock cycle.
4. **Results:** Table 6.10 shows the mean, maximum and minimum timings for each array size.
5. **Evaluation:** Both operation list building and trace compaction are fast processes. Operation list building is approximately ten times faster than trace compaction, although the exact ratio depends on the array size and the benchmark program.

Array Size	Operation List Building			Trace Compaction		
	Min.	Mean	Max.	Min.	Mean	Max.
1	2.79e+05	1.15e+06	2.86e+06	1.75e+06	6.11e+06	1.26e+07
2	2.79e+05	1.15e+06	2.86e+06	1.92e+06	6.40e+06	1.37e+07
3	2.79e+05	1.15e+06	2.86e+06	2.02e+06	6.80e+06	1.43e+07
4	2.79e+05	1.15e+06	2.86e+06	2.18e+06	7.31e+06	1.52e+07

Table 6.10: Trace build timings (in clock cycles) across all benchmark programs using four different array sizes.

Efficiency Experiment

▷ /testcases/hskit.py
Experimental measurement tool.



The longest trace compaction process required approximately 13 million clock cycles to process. This is fast on MCGREP-2 CPUs: it would take just over one second with a 10MHz clock. On a workstation with a recent x86 CPU, each microprogram generation step would complete in a matter of milliseconds.

6. **Conclusion:** The results demonstrate that the microcode generation process is efficient. Hundreds of microprograms can be generated on a workstation every second. This is ideal for testing large numbers of microprograms as part of a search process.

6.4.10 Scalability - programs of any size

Scalability is a requirement for TARGET, so that WCET reductions can be applied to any number of tasks, and tasks of any size. Previous work has achieved this by partitioning tasks into regions [240, 202, 201], with a “copy point” placed on each region boundary (section 4.3.4). In principle, MCGREP-1 CPUs supported this technique (section 5.3.7), but the support was not tested.

Support for the technique on MCGREP-2 CPUs can be demonstrated using the microprogramming interface described in section 6.2.4. The microcode generator can produce any number of microprograms, and a new set of microprograms can be uploaded at any time using the `MicroprogramUpload` function (section 6.3.5). It may also be necessary to upload the associated constant register values, but these take up less space than the microcode, so it may be possible to leave them loaded at all times.

The upload feature is demonstrated by the `jab` example program using a 3 unit array generated by MCGREP-2. The program integrates the functionality of three benchmark programs: `aes`, `blowfish`, and `jpeg`. The input for `jab` is a JPEG file which has been encrypted with both the AES and Blowfish ciphers, providing an environment in which three different sets of microcode can be used.

The IJG JPEG decoder (used by the `jpeg` benchmark [154]) allows a developer to specify an input function so that bytes of JPEG data can come from any source. This is useful for decoding

```

METHODDEF(boolean)
boolean fill_input_buffer (j_decompress_ptr cinfo)
{
    ...
    src->pub.next_input_byte = & src -> buffer [ 0 ] ;
    src->pub.bytes_in_buffer = i = fread ( src -> buffer ,
        1 , src -> bufsiz , src -> fd ) ;
    ...
    Change_Context ( aes_context ) ;
    for ( j = 0 ; j < i ; j += 16 )
    {
        /* AES decryption */
        encrypt ( & src -> buffer [ j ] ,
            & src -> temporary [ j ] , aes_ctx ) ;
    }
    Change_Context ( blowfish_context ) ;
    /* Blowfish decryption */
    BF_cfb64_encrypt ( src -> temporary , src -> buffer , i ,
        & blowfish_key , blowfish_ivec , & num , 1 ) ;
    Change_Context ( jpeg_context ) ;
    return TRUE ; /* Continue JPEG decoding */
}

```

Figure 6.20: The `fill_input_buffer` procedure uses algorithm-specific microcode for both AES and Blowfish, then loads the JPEG decoding microprogram before returning.

images as the JPEG data becomes available. Here, the input function `fill_input_buffer` is modified to pass input data through AES and Blowfish (Figure 6.20). During this operation, the RFU contents are updated by calls to the `Change_Context` procedure, loading the microcode for each encryption algorithm before the decryption function is called, then returning to the JPEG microcode before leaving `fill_input_buffer`. `Change_Context` simply calls `MicroprogramUpload` with the appropriate microprogram objects.

An interesting experiment that can be performed using this program involves variation of the block size (`src -> bufsiz` in Figure 6.20). Larger block sizes mean that each part of the program (AES, Blowfish, JPEG) does more work before handing over to the next part, which reduces the overall cost of the “upload time” taken to transfer microcode from RAM to the microprogram store as fewer transfers are needed. Figure 6.21 shows how the execution time varies with different block sizes: the execution times of each component are listed in Table B.4.

This experiment indicates that the upload cost is actually negligible in comparison to the cost of executing the encryption and JPEG processing algorithms, given a sufficiently large block size. For example, the execution time with a block size of 8192 is within 2.7% of the result with a block size of 32768. This echoes the findings of Steinke [240] and Falk [80], both of whom observed that the cost of explicitly updating a cache or scratchpad can be dwarfed by the time saved by the *use* of that cache or scratchpad, when the operation of the program can be split into partitions that have a much higher execution time than the update cost.

These results demonstrate that MCGREP-2 can support an unlimited number of regions in a specific task, and hence that scalability is possible as described in section 4.3.4. Explicitly updating

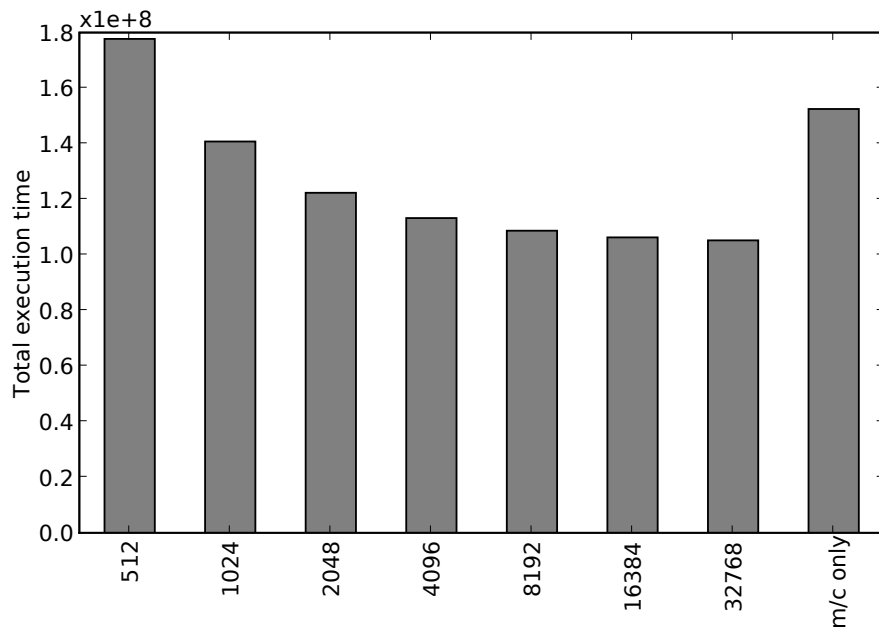


Figure 6.21: Total `jab` execution times (in clock cycles), with different block sizes. Larger block sizes allow more work to be done by each part of the program before handover to the next part. Thus, the total reconfiguration cost is reduced. (Raw data appears in Table B.4.)

Scalability Experiments

▷ /testcases/jab.py
jab experiment tool.



the microprogram store is a viable strategy for reducing execution time of large programs. The number of microprograms that can be used by a specific program is limited only by available RAM, not by the size of the microprogram store. This is not to say that the MCGREP-2 strategy for uploading microprograms could not be improved. For example, simple compression schemes implemented in hardware (such as [292]) could be used to reduce transfer times by exploiting redundancy in the microprogram encoding. Additionally, because the space taken up by each microinstruction is a whole number of words, additional time savings might be possible by packing microinstructions for multiple units together.

6.4.11 Scalability - increasing resources

Another scalability requirement for TARGET is the ability to make use of larger array sizes. Larger numbers of functional units should enable greater execution time reductions as more ILP can be exploited. There is a known limit on the total amount of ILP that is theoretically available in any program [268], but this limit should be approached by the MCGREP-2 environment within the execution of each microprogram.

Figure 6.22 shows the effects of increasing the array size on the time spent executing microcode within each benchmark program. The data in this chart has been normalised against the execution time of the machine code that is equivalent to each trace. The greatest changes result from using a custom microprogram in the first place, and introducing a second array unit. Incremental improvements are achieved by adding further units, but not for every benchmark. These results are in line with the findings of previous work: it is known that increasing the number of functional units results in diminishing returns [186, 87, 268], except in the special case of vectorisable code.

The microprogram store is also a resource of the MCGREP-2 CPU. Each program can make use of more of the microprogram store, by increasing the total size of each trace, or by marking more hotspots within each program. However, the only general way to effectively experiment with this using the MCGREP-2 tools is to manually mark more hotspots. Simply increasing the length of a trace does not necessarily reduce execution time further. In fact, it may actually increase the total execution time. This is illustrated by Figure 6.23, in which each trace length has been repeatedly increased. The results are chaotic, because of a combination of two effects:

- Greedy speculative execution. The MCGREP-2 trace-style scheduler attempts to execute every operation as early as possible. If a branch is mispredicted, the costs of this process are severe, because many operations may have been executed unnecessarily.
- Branch independence assumption. For the purposes of this evaluation, the trace generator has been guided by an execution profile in which the likelihood of each branch has been recorded based on earlier execution. Unfortunately, the profile does not capture any information about

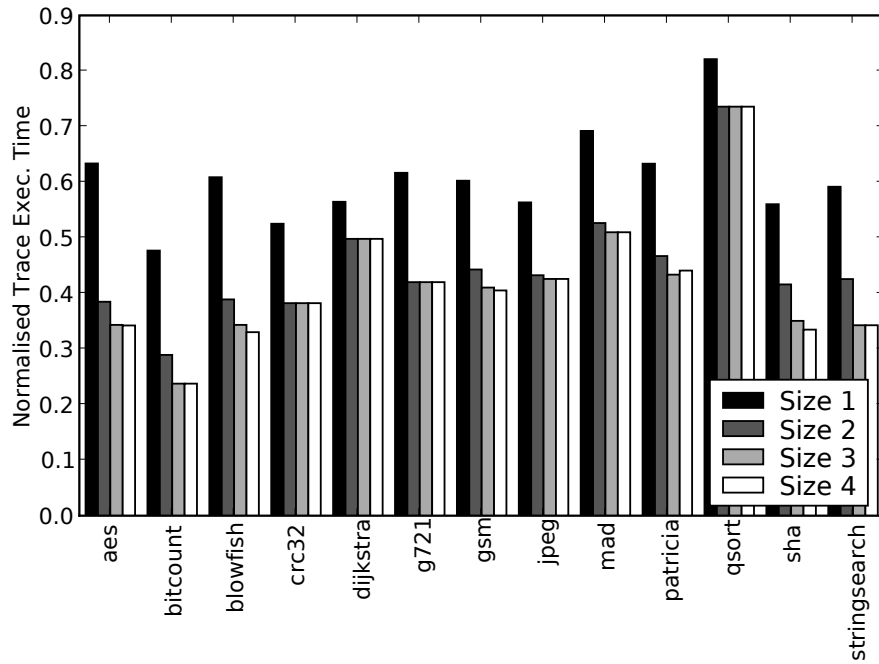


Figure 6.22: Normalised microprogram execution times within various benchmarks, with various array sizes. Each data point has been normalised against the execution time of the equivalent machine code.

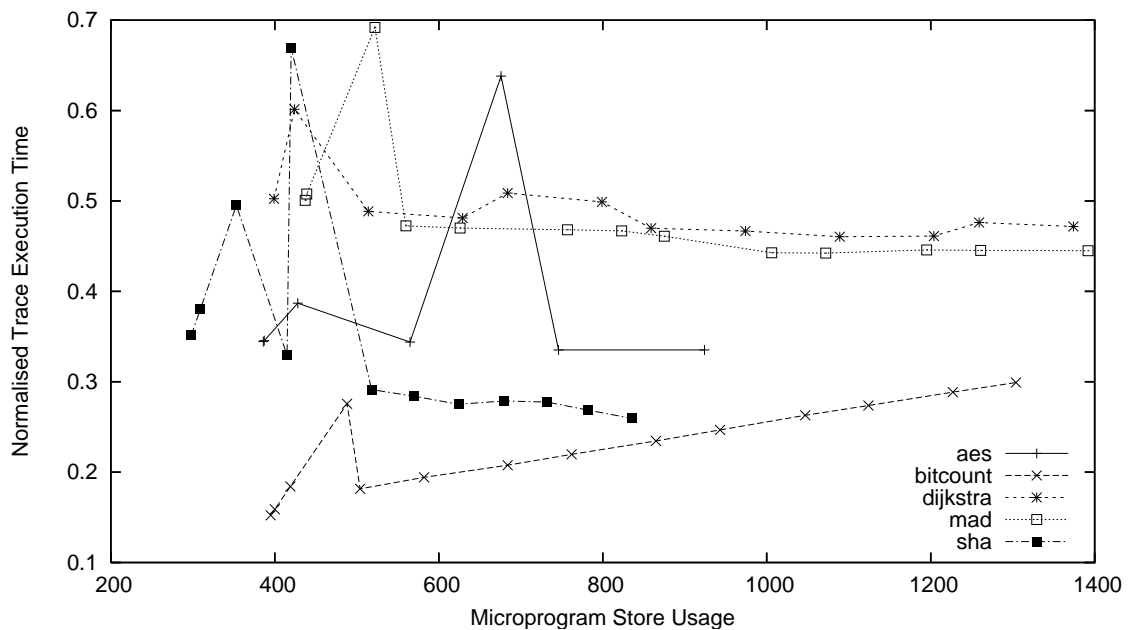


Figure 6.23: Effects of increasing trace length on normalised trace execution time of some of the benchmarks.

Scalability Experiments

▷ /testcases/hskit.py

Experimental measurement tool.



the likelihood of each branch being taken if earlier branches were also taken: in other words, there is an implicit assumption that each branch is independent of the others.

If the goal of this work were to reduce ACET, then both of these problems would need to be addressed. One important improvement would be the introduction of a *path profile* [87], which provides information about likely paths through the program rather than information about each branch in isolation.

However, while the most likely paths can be identified by path profiling, the most costly paths can generally only be found by analysis. WCET reduction approaches using the MCGREP-2 microprogram generator must be guided by two observations:

1. That increasing the length of a trace may not reduce WCET further, because there is a non-linear relationship between the length of a trace and the execution time saved by it; and
2. Introducing additional custom microprograms may be a better way to reduce WCET further.

There is a need to (1) identify the most significant contributors to the length of a WC path automatically, then (2) search for the best way to allocate microprogram store resources to reduce the length. In other words, the disadvantages of profiling can be sidestepped by a transition to the use of WCET analysis, which already provide the required features (section 4.3.5).

6.4.12 Usage of FPGA Space

The resource usage of MCGREP-2 hardware should also be considered. Table 6.11 indicates the amount of FPGA logic resources consumed by various configurations generated by MCGREP-2. All of these results were obtained by synthesising each design for a large Spartan-3 FPGA (part id `xc3s5000-fg1156-4`), with identical synthesis settings. Two different types of Xilinx optimisation were applied: “Speed” and “Area”. Debug chain features were omitted from the hardware.

The results indicate that MCGREP-2 CPU hardware has made improvements to the MCGREP-1 implementation beyond those already discussed in this evaluation. The maximum frequency is higher, and in the most directly comparable configuration (2 functional units, with one multiplier per CPU), MCGREP-2 hardware uses less memory (BRAMs) and fewer flip-flops, and only 15% more logic (LUTs). Like MCGREP-1, it uses less FPGA space than OpenRISC in this configuration, but has a lower maximum clock frequency. This is due to the different pipeline structure. The classical RISC structure of the OpenRISC pipeline allows a register to be placed between multiplexers and the ALU: this is not possible in CPUs generated by MCGREP-2, and consequently the maximum clock frequency is reduced.

Larger CPU configurations generated by MCGREP-2 hardware do not compare as favourably. The largest configuration examined (4 functional units, one multiplier per unit) is 89% larger than

Speed Optimisations					
Core	LUTs	FFs	BRAMs	Mults.	Max. Freq (MHz)
MCGREP-2: 1u	1606	187	4	3	45.4
MCGREP-2: 1u +m	1606	187	4	3	45.4
MCGREP-2: 2u	2979	188	9	3	42.6
MCGREP-2: 2u +m	3095	188	9	6	43.1
MCGREP-2: 3u	4410	190	13	3	43.5
MCGREP-2: 3u +m	4606	189	13	9	43.6
MCGREP-2: 4u	6643	190	17	3	42.4
MCGREP-2: 4u +m	6908	190	17	12	42.4
MCGREP-1	2756	228	14	3	36.8
Microblaze	1315	480	0	3	78.1
OpenRISC	3817	1095	2	3	53.0

Area Optimisations					
Core	LUTs	FFs	BRAMs	Mults.	Max. Freq (MHz)
MCGREP-2: 1u	1511	187	4	3	30.7
MCGREP-2: 1u +m	1511	187	4	3	30.7
MCGREP-2: 2u	2798	188	9	3	30.4
MCGREP-2: 2u +m	2837	188	9	6	30.4
MCGREP-2: 3u	4043	189	13	3	30.1
MCGREP-2: 3u +m	4115	189	13	9	30.1
MCGREP-2: 4u	5913	190	17	3	29.3
MCGREP-2: 4u +m	6005	190	17	12	29.4
MCGREP-1	2465	219	14	3	21.7
Microblaze	1296	473	0	3	69.7
OpenRISC	3665	1027	2	3	29.2

Table 6.11: Sizes and maximum speeds of MCGREP-2 CPU hardware, using various configurations and two different optimisation settings, with three other soft cores given for comparison. Each MCGREP-2 configuration is described by the number of functional units (e.g. 1u = 1 unit), and whether each functional unit includes a multiplier (+m). All of the cores were synthesised as for Table 5.14. (On Spartan-3 FPGAs, a 32-bit multiplier is constructed from three 18-bit multiplier modules.)

Scalability Experiments

▷ /testcases/test_various_configs.py

Synthesises various CPU configurations using MCGREP-2 (along with MCGREP-1, OpenRISC and Microblaze cores) to obtain the data in Table 6.11.



OpenRISC. The tradeoff for this extra space is better support for WCET reduction through the use of custom microprograms. OpenRISC and Microblaze do not support any form of user microprogramming: their capabilities are limited to in-order execution of machine code.

Future instances of TARGET would need to make use of the types of optimisations applied within more advanced CPUs in order to make best use of FPGA space and operate at higher clock frequencies. Some of these are already used within the Microblaze CPU, such as manual placement, manual optimisation for a particular FPGA fabric, and a higher degree of pipelining. The Microblaze CPU has essentially the same capabilities as OpenRISC, so the difference in size between OpenRISC and Microblaze indicates what is possible with manual optimisation.

6.4.13 Optimisation Possibilities

Implementations of TARGET generated by MCGREP-2 work correctly, and the microprogram generator can reduce the execution time of programs. However, many optimisations could be made in order to improve performance. Most of these involve changes to both the microcode generator and the MCGREP-2 architecture:

- The microcode generator makes a “writeback” microinstruction for each updated register at each exit, except when the updated register is written by subsequent code without a preceding read access.

These writeback microinstructions use up the microprogram store space and increase the overhead of executing each trace. More writebacks could be removed by making use of information about the ORBIS32 *application binary interface* (ABI) specified by [151], since some registers are designated as “temporary” and are not expected to be saved when function calls are carried out. Under some circumstances there is no need to update these registers at all, provided that it is possible to assume that the application conforms to the ORBIS32 ABI. However, the checkpoint mechanism (section 6.4.3) will fail a test if any register value does not match expectations. As a result, the checkpoint experiments would have to be extended with information about the ORBIS32 ABI.

The space required for writeback functionality could be reduced by implementing it using a shared microprogram. However, this would require the capability to microprogram register operations with dynamic addresses, which is not currently supported. Alternatively, a register renaming table in hardware might provide a way to eliminate the transfers entirely at the cost of additional hardware.

- Branches in traces make use of the ALU and routing resources of the first functional unit

Program	System Cycle = 1			System Cycle = 0		
	Machine Code E.T.	μc E.T.	Improvement	Machine Code E.T.	μc E.T.	Improvement
aes	5.33e+06	2.74e+06	1.94	3.36e+06	2.23e+06	1.50
bitcount	1.71e+08	1.53e+08	1.11	9.37e+07	8.75e+07	1.07
blowfish	7.14e+07	3.97e+07	1.80	4.30e+07	3.06e+07	1.41
crc32	9.27e+06	3.60e+06	2.57	5.30e+06	3.57e+06	1.49
dijkstra	5.05e+08	2.92e+08	1.73	2.86e+08	2.57e+08	1.11
g721	5.96e+08	4.82e+08	1.24	3.34e+08	3.11e+08	1.08
gsm	7.79e+07	5.83e+07	1.34	4.66e+07	3.92e+07	1.19
jpeg	1.47e+08	1.06e+08	1.39	8.91e+07	7.56e+07	1.18
mad	8.63e+07	7.84e+07	1.10	5.27e+07	5.12e+07	1.03
patricia	1.94e+08	1.46e+08	1.32	1.05e+08	9.78e+07	1.07
qsort	1.68e+07	1.59e+07	1.06	9.94e+06	9.90e+06	1.00
sha	1.24e+08	8.33e+07	1.48	7.12e+07	6.05e+07	1.18
stringsearch	1.09e+06	9.78e+05	1.12	6.51e+05	6.08e+05	1.07

Table 6.12: The effects of removing the System cycle on execution time, with and without custom microcode. Machine code operations are faster but custom RFU operations are unaffected.

in the array. This is so that the target uPC address and predicate can be obtained without additional hardware. However, this means that branches are not zero cost, because an ALU is consumed by control operations. An improvement to the control unit would allow this to be avoided.

- The microcode generator could be improved by extensions to add support for hyperblocks [170], which could allow more code to be executed by the RFU.

But the MCGREP-2 architecture could also be optimised:

- At least two clock cycles are needed for each machine code instruction execution (Table 6.8). When machine code is only executed from external RAM, this is not a bottleneck, but it can become a bottleneck when code is executed from fast scratchpad memory. This problem is due to the System microprogram, which consumes one clock cycle between every pair of machine code instruction executions.

The effects of this bottleneck are shown in Table 6.12. The results on the left show the execution time of each benchmark program on CPUs generated by MCGREP-2, with a cost of one clock cycle per System microprogram execution and minimal memory access latency. The results on the right show the execution times *as they would be* if the System microprogram had zero cost. This change reduces the time taken to execute machine code, but does not affect the time taken to execute custom microcode. Consequently, the degree of improvement made by the use of custom microcode is reduced. However, the results indicate that custom microprograms still provide an improvement even if the System microprogram is removed.

The results indicate that removing the System cycle would be worthwhile if execution is not bounded by memory latency. Pipelining CPU operations would be an effective way to do this. Suitable data path designs can be found in [195]: in general, the functionality of the System

microprogram needs to be moved into hardware that can operate in parallel with instruction execution. One possible design would migrate the System microprogram onto another unit in the RFU array, since other functional units are idle during machine code execution.

- The RFU interconnect is also not ideal for implementation in an FPGA due to the number of wires required. This problem reduces the maximum clock frequency and adds to the total size of the design (section 6.4.12). The implementation could be improved by the application of register clustering technology (section 6.3.3), and by pipelining transfers between units.

In general, the possible improvements to both the architecture and the custom microcode generator are closely related to those previously implemented by VLIW CPUs (section 2.2.2.3). But an additional condition is applied by the requirements for the work: basic block timing invariance must be maintained. While VLIW CPUs are frequently controlled by schedulers like those described in sections 6.1.2 and 6.1.3, it is assumed that code is stored in a cache [87].

A cache cannot be used in a member of the TARGET architecture. The variant of TARGET that has been explored here replaces the VLIW instruction cache with a statically programmed scratchpad: the microprogram store. To further ease the memory bottleneck, a simpler machine code can also be used to program the CPU. This can be speeded up with an instruction scratchpad [244, 273]. A high-speed ASIC implementation of a TARGET CPU would need to involve much more pipelining than the present implementation generated by MCGREP-2, but could take advantage of high-speed burst memory transactions to reload scratchpads and microprogram stores.

A VLIW-like design has been reached because of the requirement to run general machine code (section 3.2). It is not possible to translate general machine code into a fully parallelised form, such as a vector instruction [268] or an optimised hardware device [106]. This is why CGRA research has made use of custom data flow languages [101, 113] to specify hardware. VLIW CPUs avoid the problem by finding ILP in general code, and although this works well, it produces diminishing returns with increasing numbers of parallel units [268].

Arguably, the general machine code requirement is too strict. Certainly, MCGREP-2 could be extended to support ASIP-style hardware co-processors: OpenRISC has allocated opcode space for custom instructions in ORBIS32 [151]. New instructions would need to operate with well-defined timing, with functionality implemented in both C (for the simulator) and VHDL/Verilog (for hardware). But as section 2.6.2 describes, the disadvantage of this approach is reduced scalability, as limited hardware space is available for custom instructions (and co-processors in general).

The addition of a vector programming/data flow language would also improve the operation of hardware generated by MCGREP-2, because it would provide more information for the microprogram generator. More parallel microprograms would be ideal for vectorisable tasks, as Piperench has demonstrated [101]. The end result might resemble an RFU architecture such as [264].

The extensibility of the MCGREP-2 software makes it possible to implement architectural changes within an object-oriented framework. The defined interface between the microarchitecture and the microcode generator makes it possible to apply changes incrementally. Additionally, there is a complete environment for testing and debugging the components. These features facilitate improvements of any type.

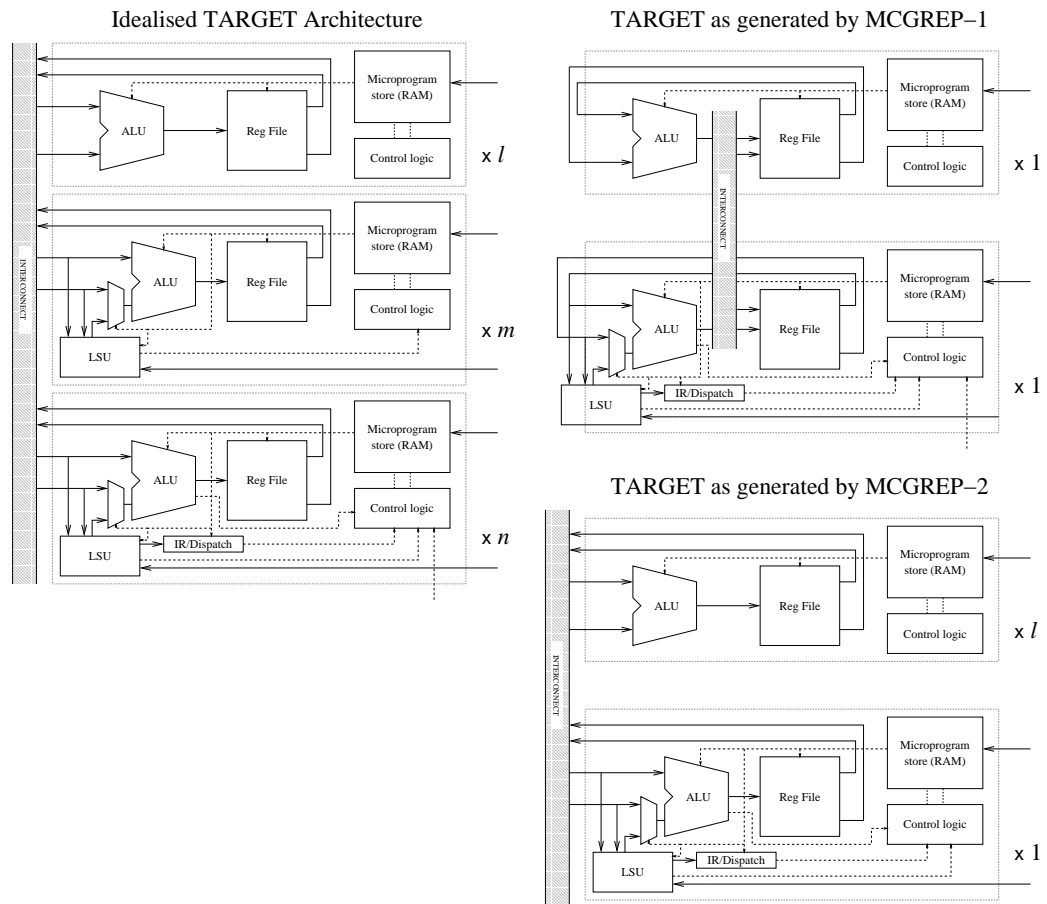


Figure 6.24: The variants of TARGET generated by MCGREP-1 and MCGREP-2, in relation to the ideal form of the architecture (left). MCGREP-2 (bottom right) allows the array size l to be configured.

6.4.14 Summary

MCGREP-2 has been shown to work correctly. It implements or generates each of the components highlighted in Figure 6.1, and generates CPUs that meet the requirements specified in section 3.2:

- Full support for general software: demonstrated by the benchmarks used (Table 6.6).
- Basic block timing invariance: demonstrated in section 6.4.7.
- WCET reduction versus previous work: demonstrated in section 6.4.8, using the assumption that each benchmark is a single-path program.
- Efficient optimisation process: demonstrated in section 6.4.9.
- Scalability: demonstrated in sections 6.4.10 and 6.4.11. Adding more units to the array can lead to greater reductions in execution time, but the profile-based strategy for identifying worst-case paths does not scale.

Figure 6.24 shows the variants of TARGET generated by MCGREP-1 and MCGREP-2. A larger subset of TARGET can be generated by MCGREP-2, which improves on MCGREP-1 through

greater scalability, support for automatic microcode generation, and improved configurability. It comes closer to the ideal implementation of TARGET, and therefore provides a better platform for reasoning about the capabilities of TARGET. Further optimisations are possible (section 6.4.13), but these are not necessary in order to make use of TARGET.

However, the issue of allocating traces for execution time reduction appears to be a complex problem. WCET reduction must involve a search process to identify the best places to use the RFU, and as section 6.4.11 describes, the search space is not simple. The next chapter applies TARGET to the problem of WCET reduction, using MCGREP-2 as part of the experimental implementation used to test a solution to this problem.

Chapter 7

WCET Reduction using IPET

The hypothesis (section 3.4) asserted that RTR hardware could be used to reduce the WCET of a program via exploitation of ILP. The TARGET architecture (chapter 4) provides the required features, and two implementations are now complete (chapters 5 and 6). With its automatic microcode generator, the MCGREP-2 implementation provides almost all of the components of the abstract WCET reduction process shown in Figure 4.1. Figure 7.1 illustrates the components that are still missing: the WCET reduction level.

In chapter 6, it was assumed that input data is fixed, and consequently the functions making the greatest contribution to the WCET could be identified by profiling. Additionally, a manual strategy was used to label hotspots (section 6.4.3), which is not scalable as the results in section 6.4.11 indicate.

Therefore, the current work is not sufficient to demonstrate the hypothesis. It has only demonstrated a reduction of ACET in the general case. What is missing is a way to identify worst-case paths in a program, then model the effects of traces produced for those paths, and finally select the best microprograms to minimise the program WCET. The techniques examined in this chapter are tested using MCGREP-2, but any instance of the TARGET architecture could be used instead because the work in this chapter is independent of any specific set of TARGET parameters (Table 4.1).

In section 7.1, a WCET reduction process is proposed based on the requirements of the hypothesis, and section 7.2 describes the implementation of this process using MCGREP-2. Section 7.3 evaluates the effectiveness of this implementation and describes possible improvements. Finally, section 7.4 describes adaptations to TARGET and the WCET reduction algorithm to support instruction scratchpads, enabling a comparison to be made between the use of custom microprograms and the use of instruction scratchpads.

7.1 WCET Reduction

Section 6.4.8 demonstrated that TARGET-based architectures can reduce the execution time of programs by using custom microcode. The single-path assumption means that this cannot be used with a general program, but WCET analysis techniques can be used to allow the same principles to be applied to any program. WCET analysis is a mature topic (section 2.1), and the literature chapter has identified IPET as the best WCET analysis approach for architectures with basic block timing invariance (section 3.1). By design, TARGET has this property.

WCET analysis has been previously applied to reduce the WCET of real-time software. Within

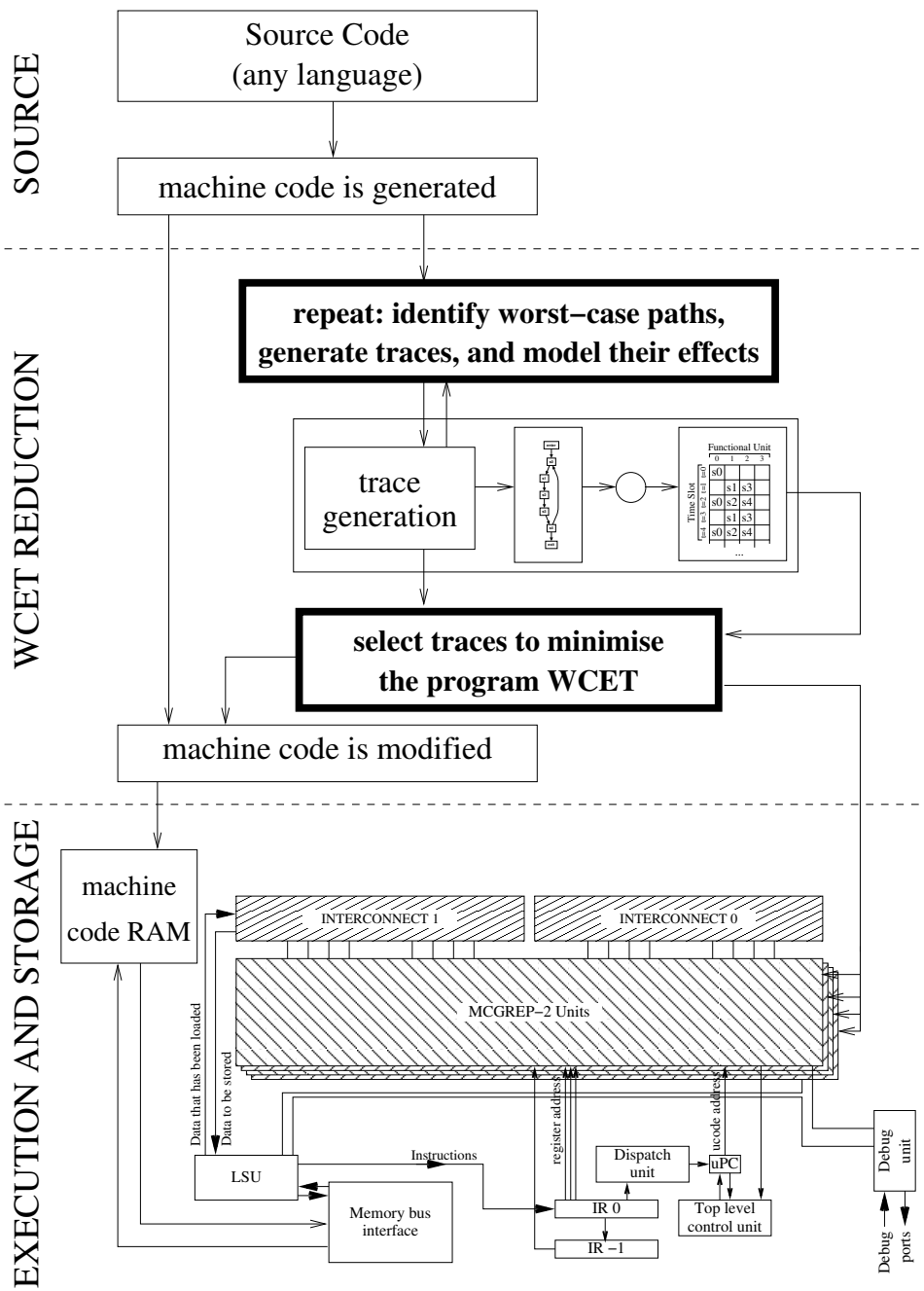


Figure 7.1: MCGREP-2 implements the execution and storage levels of the abstract WCET reduction process along with the trace generator. The other WCET reduction components (highlighted in bold) are implemented in this chapter.

a single function, Zhao et al. [295] used WC path information to guide compiler optimisations by superblock formation. On a larger scale, Wehmeyer and Marwedel [273] have applied instruction scratchpads to the WCET reduction problem, using WCET analysis to select basic blocks for migration into scratchpad memory. Their approach is effective for some examples, but it assumes that the WC path is not changed by each application. Additionally, it assumes that the scratchpad is fully loaded before execution of the program begins: it is not updated during execution.

An improvement has been made to the process by Suhendra et al. [244], with an algorithm that considers the effect of each migration on the WCET (Figure 2.9). This improves the results, but it is still assumed that the scratchpad is loaded before execution starts. Falk et al. [80] describe a similar WCET reduction algorithm using a locked instruction cache in place of a scratchpad.

Suhendra's algorithm is improved further by Puaut and Pais [202, 203], with extensions to support the analysis of programs that are divided into overlay regions (section 2.3.12). The Puaut and Pais algorithm (Figure 2.10) considers scratchpad loading time as a part of the WCET reduction problem rather than simply assuming that it takes place before execution. It can also carry out several migrations per WCET evaluation, which results in a lower running time with a possible loss of accuracy. This approach requires a program to be partitioned beforehand, using a partitioning algorithm such as [201] or the manual specification of partition boundaries [240].

The work in this chapter follows the scratchpad loading assumption made by Wehmeyer, Suhendra and Falk, who consider scratchpad (or cache) loading to be a separate process that is distinct from execution. This has two consequences: (1) the loading time is not considered, and (2) partitioning is not considered. These assumptions make it possible to concentrate on WCET reduction without considering partitioning issues. Extensions for partitioning are outside the scope of this work, but some experiments are described in Appendix G.

There are five general subproblems in WCET reduction, two of which have already been implemented by TARGET:

1. **Building an architecture for WCET reduction:** (implemented, section 6.2)

This subproblem has already been addressed by TARGET and the MCGREP-2 implementation. Previous work [273, 80] has made use of existing CPU architectures, which are ideal for scratchpad and locked cache approaches, but not suitable for the vertical migration approach that enables ILP to be exploited across basic block boundaries, which is possible using TARGET.

2. **Identifying the WC path:** (not implemented yet)

In [273] and [80], this is done using the `aiT` WCET analysis tool. A worst-case path makes a significant contribution to the overall WCET, so the WC path can be identified by looking for parts of the program with large local WCET values (section 4.3.5). `aiT` has been used by [80] because it is able to model the CPU under consideration (an ARM920T). IPET-based methods could be adapted to model CPUs with similar features [160], but the integer linear program might become too complex to solve: this is why Wilhelm [280] advocates combining abstract interpretation and integer linear program models for complex CPUs, as implemented in `aiT`. However, this problem is not applicable to TARGET because of its basic block timing invariance property, so IPET can be used. The same is true of the CPU architecture used for scratchpad allocation evaluation by Puaut [202].

3. **Reducing the cost of the WC path:** (implemented, section 6.3)

In [273, 202], the costs of worst-case paths are reduced by migrating code to instruction scratchpad. Suhendra [244] considers migrating both instructions and variables. Falk [80] migrates code to a locked instruction cache.

This WCET reduction subproblem has already been solved by the microcode generator described in section 6.3, which applies trace-style scheduling to a sequence of basic blocks to reduce their execution time. Building traces along a subsequence of the WC path will reduce the WCET of a program. The microprogram store of TARGET is similar to an instruction scratchpad in that executable code is allocated to low latency RAM, close to the CPU, but further reductions are possible due to the exploitation of ILP across basic block boundaries. This approach is related to that of Zhao [295], which also makes use of the formation of superblocks along the WC path, but within the context of compiler optimisations for a single function rather than reducing the WCET of an entire program.

4. **Modelling the effects of each WCET reduction:** (not implemented yet)

[244], [202] and [80] all reevaluate the WCET of a program after each WCET reduction is applied. This is necessary (1) because each WCET reduction may change the WC path, and (2) because the final WCET must be known in order to be certain that the program is safe (section 2.1). The process is performed using a WCET analysis tool.

When code is being migrated into an instruction scratchpad or locked cache, it is quite easy to determine both the scratchpad space cost and the reduction in execution time cost for that code. Shifting code into a scratchpad just reduces the memory latency when that code is executed. The space cost is the size of the code.

The benefits of a custom microprogram are not so easily identified by looking at the input machine code, because of the complexity of the transformations applied by the microprogram generator. The microprogram must be executed to obtain the execution time of each path.

5. **Allocating limited resources to minimise WCET:** (not implemented yet)

Scratchpad or locked cache space is limited, so selecting code for migration is a form of optimisation problem. For reductions of ACET and energy usage, the 0-1 knapsack problem provides an ideal model for this purpose [240, 241]. For WCET reductions, more complex models are needed [244, 80]. However, the same approaches that apply to instruction scratchpads and locked caches are applicable to allocating space in the TARGET microprogram store.

Since the problem of implementing TARGET has already been addressed by MCGREP-2, and the problem of reducing the execution time of a path through machine code has already been addressed by a trace-style scheduler (section 6.3), the following three problems remain to be solved:

- Identifying the WC path (section 7.1.1),
- Modelling the effects of each possible WCET reduction (sections 7.1.2 to 7.1.8),
- Allocating the limited space within the microprogram store to minimise the WCET (sections 7.1.9 to 7.1.12).

The discussion of these problems is followed by a summary in section 7.1.13.

7.1.1 Identifying Worst-Case Paths

The IPET approach for WCET analysis [159] provides two things: a method to determine the WCET of a program, and a way to find the parts of that program that make the greatest contribution to that WCET. The Puschner and Schedl IPET model [207] is used because (1) it accounts for looped code using implicating edges and relative capacity constraints, and (2) a proof of correctness exists.

The Puschner and Schedl IPET model has been previously discussed in section 2.1.2, but because the model needs to be extended here, a more detailed examination is necessary. A worked example of the IPET process is given in Appendix E. Summarising, the model is composed of the following parts:

- A program is represented by a T-graph $G = (V, E)$.
- Every edge $x \in E$ represents a (possibly empty) basic block $b(x)$.
- Every vertex $v \in V$ represents a point between two or more basic blocks. Vertices may represent control flow splits (e.g. `if` statements) or join points.
- The execution time cost of each edge $x \in E$ is defined as $\gamma(x)$. Each $\gamma(x)$ is a constant positive integer, such as a CPU clock cycle count.
- The number of times each edge $x \in E$ is executed as part of the WC path through the code is defined as $f(x)$. The WC path is the path that maximises the execution time.
- The WCET of program G is defined by an equation:

$$Z_G = \sum_{x \in E} \gamma(x) f(x) \quad (7.1)$$

An integer linear program is used to maximise Z_G by selecting appropriate values of $f(x)$. In this work, $f(x)$ is referred to as the *worst-case flow* (WC flow). It represents the maximum number of executions of each edge x along the WC path through the program. The values of $f(x)$ are bounded by the following types of constraint:

- **Conservation of flow constraints:** execution flow is conserved at each vertex, i.e.

$$\forall v_1 \in V, \sum_{v_0 \in V} f(v_0, v_1) = \sum_{v_2 \in V} f(v_1, v_2) \quad (7.2)$$

To avoid needing to treat the entrance and exit of the program as special cases where flow is not balanced, Puschner and Schedl use an additional back edge to link them. The flow through this back edge is always one. The execution time cost is always zero.

- **Relative capacity constraints:** the flow through a loop is relative to the flow in an edge leading into that loop. This is defined by a relation between two flow values, such as $f(x) \leq 10f(y)$. These constraints are needed because any flow could otherwise exist within a loop without violating equation 7.2.
- **Developer-specified constraints:** such as those listed in Table 2.1. These provide information about possible paths through the program, including cases such as the example of Figure 2.3 where some paths are infeasible. All of these are translated into a linear relationship between two or more flow values.

A side effect of maximising Z_G is that WC flow values are assigned to each $f(x)$. As previously noted in section 4.3.5, this makes it possible to identify the greatest contributors to the WCET. These are basic blocks that make the greatest contribution $\gamma(x)f(x)$ to the overall WCET Z_G (equation 7.1).

Previous WCET reduction approaches have made use of WC flow values (as defined above) to identify candidates for migration into instruction scratchpads [273, 244, 202] or locked caches [80]. But modelling the effects of these migrations on WCET is relatively simple, because the only effect of each change is the reduction of one basic block cost $\gamma(x)$ due to the new memory latency. Because traces span multiple basic blocks, more complex models are required.

7.1.2 Modelling the Effects of a WCET Reduction

The effects of introducing custom microcode (i.e. a trace) need to be incorporated into the IPET model. The principles of this process have been outlined in section 4.3.5: adding a trace to a program effectively transforms the T-graph, changing the execution time costs of some paths, potentially creating a new WC path, and (if the trace is added correctly) reducing the WCET.

The existing IPET model is composed of the equations, variables and constraints listed in the previous section. The model must have the following properties:

- Equivalent to the Puschner and Schedl IPET model when no traces are in use.
- Preserves the correctness of the Puschner and Schedl IPET model.
- Able to model any trace that could be generated by the software described in section 6.3. This includes traces that include many copies of the same basic blocks, a common occurrence within loops.
- Able to model multiple traces in a single program, even if the traces have basic blocks in common.

The new model will extend the Puschner and Schedl model. The extension must have two components: (1) a model of the sequence of functions carried out by a trace, and (2) a model of the flow through a trace. The first part provides a way to relate traces to machine code, based on the observation that traces are functionally equivalent to the basic blocks that they are derived from. This is described in section 7.1.3. The second part provides a way to incorporate knowledge about the WC flow through a trace. This is described in section 7.1.4.

Multiple traces are addressed in section 7.1.5, and the issue of constraints is examined in section 7.1.6. Equivalence to Puschner and Schedl when no traces are in use is demonstrated in section 7.1.7, and the application of a proof based on Puschner and Schedl's work is discussed in section 7.1.8.

7.1.3 Modelling the Functionality of a Trace

The defining property of a trace is the sequence of T-graph edges that make it up. In turn, this effectively depends on the starting "entrance" edge of the trace, $e \in E$. The actual algorithm that selects successive edges does not have to be modelled since it is sufficient to model its effects, i.e. the paths it creates and their execution time costs.

A trace beginning at edge e is labelled T_e . T_e is defined as the set of all paths through traces beginning with edge e . There is exactly one path per exit. For example, the trace shown in Figure

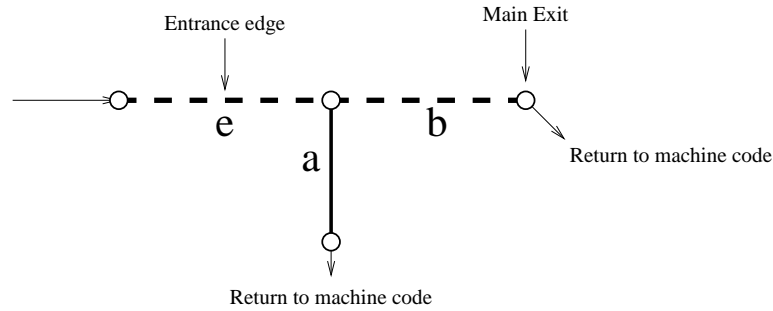


Figure 7.2: A trace as part of a T-graph, with two exits.

7.2 includes three T-graph edges and has two exits. The possible paths through this trace are $P_{e,1} = [e, a]$ and $P_{e,2} = [e, b]$, so this trace is defined as $T_e = \{P_{e,1}, P_{e,2}\}$. Generally, any path is a sequence of edges:

$$P_{e,i} = [e, e_1, e_2, \dots, e_n] \quad (7.3)$$

and any trace T_e with m exits is a set of m paths beginning at e :

$$T_e = \{P_{e,1}, \dots, P_{e,m}\} \quad (7.4)$$

The notation $P_{e,i,j}$ refers to the j -th member of path $P_{e,i}$, which is in turn the i -th path beginning at edge e . Representing traces by the paths that they contain is useful for four reasons:

1. Any execution of a trace T_e must run through the whole of exactly one of the paths $P \in T_e$. This allows the WC flow through each trace path to be modelled, just as the WC flow through each basic block is modelled by the original IPET model.
2. The Puschner and Schedl IPET model makes use of paths in an inductive proof of correctness, in which it is shown that (1) the WCET is correctly calculated when there is one path through the program, and (2) that if the WCET can be correctly calculated for n paths, then adding a new path will preserve correctness. Since the paths within a trace are subsequences of paths through the entire program, paths are neither created nor destroyed by the addition of a trace. Transforming the execution time of paths (without creating new ones) does not prevent the Puschner and Schedl proof being reapplied to the new model.
3. The path model can accommodate paths that pass over the same basic blocks repeatedly. This is necessary to handle traces that might be formed in loops.
4. The path model does not prevent multiple traces passing through the same basic block, which is useful for tail duplication and other places where multiple traces meet.

The contents of each path P and the associated execution time cost $\gamma(P)$ can be determined from the output of the trace generator. This information is a side effect of the trace generation process.

7.1.4 Modelling Worst-Case Flow of Traces

Paths provide a way to incorporate information about the actions of the trace generator into the IPET model. But more is needed so that the WC flow values f can be determined for parts of the

program, whether they are in machine code, or in a trace, or both. Let Θ be the set of all traces in a program $G = (V, E)$.

To do this, the notion of *contexts* is introduced. Each edge $x \in E$ within the T-graph represents a basic block $b(x)$. More generally, each edge x represents some functionality. That functionality is also part of any trace path that includes x : this is necessary so that traces can be functionally equivalent to original machine code.

A context of x is any place in which the functionality of x may execute. All of the contexts of x are represented by the set $c(x)$. At a minimum, each $c(x)$ includes $u(x)$, which represents the execution of the original machine code:

$$\forall x \in E, u(x) \in c(x) \quad (7.5)$$

u symbolises untraced execution. There is an additional context for every occurrence of x within a trace T_e , represented by $t(P_{e,i}, j)$. (This is the j -th member of path i of trace T_e .) It is possible to formally define $c(x)$ as follows:

$$\forall x \in E, c(x) = \{u(x)\} \cup \{t(P_{e,i}, j) : T_e \in \Theta \wedge P_{e,i} \in T_e \wedge x = P_{e,i,j}\} \quad (7.6)$$

The purpose of this distinction is to allow flow to be considered separately for machine code execution (i.e. $f(u(x))$) and for execution in any trace (i.e. $f(t(P_{e,i}, j))$). In effect, a new flow variable is created for each $x \in E$ and each context in $y \in c(x)$.

In the Puschner and Schedl IPET model, the total worst-case flow through x is defined as $f(x)$. This definition is retained, but now the WC flow through traces and machine code is separated into distinct symbols within $c(x)$. The relation between $f(x)$ and $c(x)$ is:

$$\forall x \in E, f(x) = \sum_{y \in c(x)} f(y) \quad (7.7)$$

If $c(x) = \{u(x)\}$, then $f(u(x)) = f(x)$: i.e. edges that are not represented in traces are unaffected by traces. Relationships also need to be defined between new flow variables. Flow within a trace path is conserved:

$$\forall \alpha, \forall \beta, f(t(P_{e,i}, \alpha)) = f(t(P_{e,i}, \beta)) \quad (7.8)$$

Flow can only enter a trace path $P_{e,i}$ through the entrance e , where machine code flow is converted into traced flow:

$$\forall (v_1, v_2) \in E, \sum_i f(t(P_{(v_1, v_2), i}, 1)) = \sum_{(v_0, v_1) \in E} f(u((v_0, v_1))) \quad (7.9)$$

Flow can only leave a trace path $P_{e,i}$ through its exit, which is the final member of the sequence, $P_{e,i, |P_{e,i}|}$. Flow then returns to machine code:

$$\forall (v_0, v_1) \in E, \sum_i f(t(P_{(v_0, v_1), i}, |P_{(v_0, v_1), i}|)) = \sum_{(v_1, v_2) \in E} f(u((v_1, v_2))) \quad (7.10)$$

Additionally, a trace T_e always replaces machine code execution in the entrance edge e :

$$\forall e \in E, \exists T_e \implies f(u(e)) = 0 \quad (7.11)$$

As in the original Puschner and Schedl model, flow is also conserved within machine code execution:

$$\forall v_1 \in V, \sum_{(v_0, v_1) \in E} f(u((v_0, v_1))) = \sum_{(v_1, v_2) \in E} f(u((v_1, v_2))) \quad (7.12)$$

Equations 7.6 through 7.12 define the new flow variables required to represent any set of traces. However, the execution time cost of each path is still missing. Just as $f(u(x))$ represents the WC flow through edge x in one context, $\gamma(u(x))$ represents the execution time cost of that context. Each $\gamma(u(x))$ can be directly derived from the Puschner and Schedl model:

$$\forall e \in E, \gamma(u(x)) = \gamma(x) \quad (7.13)$$

Each $\gamma(t(P_{e,i}, j))$ can be derived from the total execution time cost of the path $\gamma(P_{e,i})$, which is produced as a side effect of trace generation:

$$\gamma(P_{e,i}) = \sum_j \gamma(t(P_{e,i}, j)) \quad (7.14)$$

It is possible to derive each $\gamma(t(P_{e,i}, j))$ using a series of simultaneous equations: one for each path in T_e . There may be many possible solutions to these equations, but all of them are equivalent, because only the *total* execution time cost of each path is important. Costs can be distributed to edges in any arrangement that satisfies this. The only other constraint on each value is that no cost can be negative, i.e.: $\forall \alpha, \gamma(\alpha) \geq 0$.

7.1.5 Multiple Traces and the WCET

The model described in the previous sections is sufficient to allow zero or more traces to be modelled in one program, but it is useful to introduce new terminology to allow multiple traces to be described. In the Puschner and Schedl IPET model, a T-graph G represents a program. The WCET of G could be represented as Z_G (as in equation 7.1). Let Z_{G_Θ} represent the program G plus the set of traces Θ . The definition of Z_{G_Θ} is:

$$Z_{G_\Theta} = \sum_{x \in E} \sum_{y \in c(x)} f(y) \gamma(y) \quad (7.15)$$

This is the new objective function for IPET analysis of a program G with zero or more traces.

7.1.6 Constraints

Section 7.1.1 describes three types of constraint that apply to IPET models. Of these, the conservation of flow constraints are incorporated into the new model by equations 7.9 through 7.12. Relative capacity constraints and developer specified constraints are still captured within the new model by the continued presence of $f(x)$ for each $x \in E$. In other words, although execution through any edge x may be carried by machine code or one or more traces, the total flow through that edge is still constrained by rules affecting each $f(x)$.

7.1.7 Equivalence to the Puschner and Schedl model

The new model is equivalent to the Puschner and Schedl model when $\Theta = \emptyset$, i.e. no traces are present. From equation 7.6, $c(x) = \{u(x)\}$ when no traces are in use. In this circumstance,

equation 7.7 indicates that $f(u(x)) = f(x)$ and equation 7.13 indicates that $\gamma(u(x)) = \gamma(x)$. Consequently, the conservation of flow rules in equation 7.12 are redundant, since they are already expressed by the existing IPET rules for each $f(x)$. In other words, the constraints are unchanged.

The WCET definition of equation 7.15 is equivalent to the original definition (equation 7.1), since:

$$\sum_{y \in c(x)} f(y) = f(u(x)) = f(x) \quad (7.16)$$

and

$$\sum_{y \in c(x)} \gamma(y) = \gamma(u(x)) = \gamma(x) \quad (7.17)$$

The other equations in section 7.1.4 do not apply at all, since no traces are present ($\Theta = \emptyset$).

7.1.8 Adapting the Puschner and Schedl proof

Puschner and Schedl's proof remains applicable to the new model, even when $\Theta \neq \emptyset$. The original proof shows that the WCET Z_G can be correctly computed if all program behaviour constraints are specified and all execution costs (γ values) are known. The definitions in sections 7.1.2 through 7.1.7 cover these constraints and specify the execution costs.

Puschner and Schedl's proof makes use of global execution paths. Like paths within a trace, these are sequences of edges, but each path spans the entire program. The first member of each path is the program entry point, and the final member is the program exit point. The proof has two claims: (1) if there is exactly one path $P_1 = (x_{1,1}, \dots, x_{1,n})$, then Z_G can be correctly computed; and (2) if there are n paths $\{P_1, \dots, P_n\}$ and Z_G can be correctly computed, then adding a new path P_{n+1} preserves the correctness of Z_G .

Traces do not add new paths. Instead, they replace subsequences of existing paths with functionally equivalent code that executes in less time. Because of this, and because the complete set of constraints and execution costs are known, the original proof is still applicable to the new model.

7.1.9 Allocating the Microprogram Store Space

The preceding sections describe the properties of an implementation of an integer linear program to maximise Z_{G_Θ} : the WCET of a program G plus a set of traces Θ . This is an essential part of a search process to find the best Θ for a specific program and a specific TARGET instance. The best Θ must (1) fit within the microprogram store space and (2) minimise the program WCET.

For ACET reduction and energy usage minimisation, this problem has been previously solved as an instance of the 0-1 knapsack problem [240, 241]. Generally, the inputs for the 0-1 knapsack problem are a collection of n items, each with a cost C_i and a benefit B_i . The problem aims to select a subset of these items such that the total benefit is maximised while the total cost remains within a limit C_{max} . The knapsack problem may be solved by an integer linear program. In the specific case of scratchpad allocation, C_{max} is the total scratchpad space, C_i is the space taken up by item i , and B_i is the ACET or energy reduction resulting from the selection of item i .

However, the 0-1 knapsack problem is not ideal for WCET reduction because it assumes that reductions are independent. In fact, each WCET reduction may change the WC path through the program. Failing to account for this is a known issue with Wehmeyer's scratchpad-based WCET reduction approach [273], identified by Suhendra [244], Puaut [202] and Falk [80].

Suhendra’s scratchpad allocation algorithms for WCET reduction [244] determine a subset of variables (or basic blocks) to be migrated from RAM to a scratchpad. The problem is expressed as a decision tree in which each non-leaf node represents an allocation decision: migrate object X into scratchpad, or leave X in RAM. Each algorithm traverses this tree in a depth-first order and evaluates the WCET when a leaf node is reached. Leaf nodes represent the points at which no further decisions can be made.

The first algorithm is optimal, but may involve an $O(2^n)$ exhaustive search. The search space is cut down by using a heuristic function to calculate the upper bound on a WCET reduction that is possible in any subtree. If a greater WCET reduction is possible at any of the leaf nodes found so far, that subtree can be pruned. The heuristic function proposed by [244] is a 0-1 knapsack problem, used to determine the greatest WCET reduction that is possible if independence between allocations is assumed. (This step is equivalent to the algorithm applied by Wehmeyer for WCET reduction [273].) Thus, although Wehmeyer’s approach is non-optimal, it is useful in pruning the search space.

The second algorithm (Figure 2.9) is described as a “greedy heuristic” and can be used on larger problems due to a lower time complexity. It repeatedly obtains the cost of each part of the current WC path, then allocates the object X making the greatest contribution to the WCET. This process continues until scratchpad space is exhausted or no further candidates can be identified. Suhendra et al. report that this approach is almost as good as the optimal algorithm in most cases.

Puaut and Pais [202, 203] describe a scratchpad allocation algorithm with much in common with Suhendra’s “greedy heuristic” (Figure 2.10). Two improvements are made over Suhendra: (1) the new algorithm supports partitioning (programs that are divided into multiple regions, each with a separate scratchpad memory map), and (2) more than one allocation decision can be made between each WCET evaluation. The first improvement is not considered in this chapter (section 7.1). The second improvement can speed up the allocation process, since WCET analysis is the slowest step of Suhendra’s algorithm, but the results may be no better than Suhendra’s algorithm because WCET reduction decisions are being made with less information about the WC path.

Falk’s locked instruction cache allocation algorithm for WCET reduction [80] also follows the Suhendra and Puaut approach by determining a subset of basic blocks to be migrated to scratchpad. Falk’s approach is also iterative, with objects being selected from the WC path and evaluated to find the greatest contributor to the WCET. A new WC path is identified after each allocation, but this is done using a partial WCET analysis rather than a complete reanalysis. The partial analysis only examines the parts of the program that have been affected by the allocation, which saves time by reusing the information obtained during previous WCET analyses.

The effects of introducing a trace are not the same as the effects of moving code to a scratchpad or locked cache, so none of these algorithms are directly applicable. But the algorithms described by Falk, Suhendra and Puaut are all designed to work quickly, avoiding searches over exponential space, and they are a good match for the requirements of the allocation problem. Regardless of whether the target is a microprogram store or a scratchpad, the same problem exists: limited memory resources need to be allocated to minimise WCET. The most important features of all of the algorithms are:

- On each iteration, the WC path is identified.
- Candidate objects are selected from the WC path and evaluated. The WCET reduction from migrating each object is computed, and the best option is selected.

- The WCET is recomputed after each assignment is made.

The results of Falk and Suhendra indicate that strategies involving these features can be successful. The results of Puaut show that the strategies can be extended to support partitioning. The model described by sections 7.1.2 to 7.1.8 provides a way to compute the effects on WCET of custom microcode. The best candidates for execution time reduction can be identified using the $f(x)$ values from the IPET model: a similar strategy is used by Falk, Puaut, Suhendra and Wehmeyer. Previous results suggest that allocation strategies are unlikely to be optimal, because an optimal allocation involves an exhaustive search [244], but nevertheless it is possible to approach an optimal allocation using a fast heuristic.

7.1.10 Microprogram Store Space Allocation Algorithm

In the following sections, a design for a WCET reduction algorithm is described and justified. The design is based on Suhendra's heuristic for instruction scratchpad allocation [244]. In the new algorithm:

- The WC path is identified,
- Various microprograms are tested as optimisations for the WC path, and the WCET reduction of each is evaluated,
- The best trace is chosen and added to the program,
- The process repeats.

Suhendra's heuristic is chosen as a base because it is the simplest allocation algorithm that is known to be effective. (The Puaut and Pais algorithm is slightly more complex because it includes support for partitioning and load time modelling.) However, the original algorithm is not useful for allocating microprogram store space because microprograms as described in section 6.3 are characterised by two properties: starting point and path length. This adds another dimension to the search problem. Since there is a tradeoff between length and microprogram store space, the space used for optimising one part of the WC path might have a greater effect than the space used for optimising another part. So the allocation process needs to (1) select good starting points for custom microcode, then (2) select good lengths for the path implemented by each microprogram.

These problems cannot be solved simultaneously, because doing so will commit microprogram store space to a specific set of microprograms without any knowledge of the space requirements of subsequent microprograms. Therefore, Suhendra's heuristic needs to be extended with a second phase in which lengths are selected.

The two phases of the allocation algorithm are discussed in the next two sections. The first (identifying the WC path and evaluating microprograms) is examined in section 7.1.11. The second (selecting microprograms) is examined in section 7.1.12. A worked example of the application of the algorithm to a simple program is also given in Appendix F.

7.1.11 Phase 1: Selecting Good Microprogram Starting Points and Evaluating Microprograms

In a large program, there might be many thousands of possible start points for custom microprograms, each with a wide range of possible lengths. For example, a loop body that is executed 100

times could accommodate a custom microprogram spanning 1 to 100 basic blocks. The search space needs to be cut down.

It seems logical to regard the allocation problem as a three-dimensional search. The first dimension is composed of allocation decisions that have already been made, as in Suhendra's heuristic. The second dimension is composed of possible microprogram start points, given the existing allocation decisions. The third dimension is composed of possible lengths for the trace path implemented by each microprogram. It is possible to prune this potentially vast 3D search space by considering only the most likely candidates in each dimension, i.e.:

- The T-graph edges making the greatest contribution to overall WCET (for the first and second dimensions). Other edges are unlikely to be useful starting points for any microprogram. The limitations on the first and second dimensions are given the symbols H and W respectively.
- The lengths below a certain fixed limit (for the third dimension). Very long trace paths will not fit in the microprogram store when their machine code is converted to microcode. This limit is given the symbol L .

At each point in the first dimension, the second dimension (starting points) should be searched to find the best place to add a trace of *any* length. This provides a *surrogate trace* that represents a trace with a specific length, allowing new WC paths to be identified in subsequent iterations. Once this surrogate trace has been identified, *candidate traces* of different lengths with the same start point can also be evaluated to obtain their WCET reductions and space costs. This information is used by the second phase to pick out the custom microprograms that make the best tradeoff between WCET reduction and space usage.

The first phase of algorithm must identify the WC path. This is done using two heuristics:

- *Compute_Scores* (Figure 7.3) produces an estimate of the maximum possible WCET reduction along the WC path beginning at e . This allows start points to be sorted in order of their estimated usefulness: the score. Each score depends upon the total $f(x)\gamma(x)$ for every edge x in the WC path beginning at e .
- *Find_WC_Path* (Figure 7.4) builds a worst-case path starting at edge $e \in E$. The path follows the WC flow from that edge onwards, based on information obtained from IPET analysis. At split points, the direction that is taken is the direction with the greatest WC flow value.

Having found the WC path, the algorithm proceeds as shown in Figure 7.5. The following operations are executed H times to make up to H possible allocation decisions. Each decision (1) identifies the basic blocks that currently make the greatest contribution to WCET using the score heuristic, then (2) finds the code to be optimised by running the *Find_WC_Path* function at each of the W greatest estimated contributors to the WCET, assuming that the path length l is maximised ($l = L$). (3) A custom microprogram is generated for each of these paths, and (4) the WCET reduction benefit of that microprogram is computed. (5) The algorithm chooses the trace starting point e with the greatest benefit B_e .

Next, the algorithm (6) evaluates the cost and benefit of traces with path lengths from 1 to L beginning at e . These are the *candidate traces*. The candidate with the highest benefit is found (7) then added to the program (8). This is the *surrogate trace*. This final step ensures that subsequent iterations will take representatives of the chosen traces into account, enabling a new WC path to be found after each reduction.

```

procedure Compute_Scores( $G$ ):
  ( $V, E$ ) =  $G$ 
  for  $e \in E$ :
     $P_e = \text{Find\_WC\_Path}(G, e, e, \infty)$ 
     $e.\text{dom\_cost} = \sum_{x \in P_e} \gamma(x)$ 
  end for
  for  $e \in E$ :
     $P_e = \text{Find\_WC\_Path}(G, e, e, \infty)$ 
     $e.\text{score} = \sum_{x \in P_e} x.\text{dom\_cost} \times f(x)$ 
  end for
end procedure

```

Figure 7.3: Heuristic to evaluate the score of each basic block: this is an indication of its contribution to WCET.

```

function Find_WC_Path( $G, e_1, e_n, l$ ):
  if  $l \neq 0 \wedge e_1 \neq e_n$ :
    ( $V, E$ ) =  $G$ 
    ( $v_0, v_1$ ) =  $e_1$ 
    if  $|\{v_2 : (v_1, v_2) \in E\}| > 1$ :
      //  $e_1$  is followed by a branch
       $l = l - 1$ 
    end if
    for ( $v_1, v_2$ )  $\in E$ :
      if  $f((v_1, v_2)) > \frac{1}{2}f(e_1)$ :
        return [ $e_1$ ] + Find_WC_Path( $G, (v_1, v_2), e_n, l$ )
      end if
    end for
  end if
  return [ $e_1$ ]
end function

```

Figure 7.4: Heuristic to find a subsequence of basic blocks within the WC path, beginning at basic block e_1 and ending at either e_n or after l branch points.

```

procedure Find_Candidates( $G$ ):
  invariant  $G$  = program as T-graph [207]
  constant  $H$  = max. number of WC paths to find
  constant  $L$  = longest trace to consider
  constant  $W$  = number of starting points to consider per step
   $\Theta = \emptyset$ 
   $D = []$ 
  for  $i$  from 1 to  $H$ :
     $Z_0 = \text{Calculate\_WCET}(G_\Theta)$  // see sections 7.1.2 through 7.1.8
     $(V, E) = G$  // members of  $E$  represent basic blocks
    Compute_Scores( $G$ ) // see Figure 7.3
    (1) sort  $E$  in order of  $e$ .score
    for  $j$  from 1 to  $W$ :
      repeat:
         $e = \text{pop}(E)$  // get "highest scoring" block in  $E$ 
        until  $e$  is a valid start point for a trace
        (2)  $P_e = \text{Find\_WC\_Path}(G, e, 0, L)$  // see Figure 7.4
        (3)  $T_e = \text{Generate\_Trace}(P_e)$  // see section 6.3
         $Z_e = \text{Calculate\_WCET}(G_{\Theta \cup \{T_e\}})$ 
        (4)  $B_e = Z_0 - Z_e$  // find benefit of this trace
      end for
    (5) find  $e$  such that  $B_e$  is maximised
    for  $k$  from 1 to  $L$ :
       $P_{e,k} = \text{Find\_WC\_Path}(G, e, 0, k)$ 
       $T_{e,k} = \text{Generate\_Trace}(P_{e,k})$ 
       $Z_{e,k} = \text{Calculate\_WCET}(G_{\Theta \cup \{T_{e,k}\}})$ 
      (6)  $C_{e,k} = \text{space cost of trace } T_{e,k}$ 
       $B_{e,k} = Z_0 - Z_{e,k}$  // benefit of this trace
      if  $B_{e,k} > B_e$  then:
        (7)  $B_e = B_{e,k}$ 
            $T_e = T_{e,k}$ 
      end if
    end for
    (8)  $\Theta = \Theta \cup \{T_e\}$ 
        $D = D + [e]$ 
  end for
end procedure

```

Figure 7.5: The *Find_Candidates* procedure finds and evaluates candidate traces.

Thus, the algorithm makes up to H allocation decisions. For each, W possible trace start points are tested, and path lengths ranging from 1 to L are evaluated. The heuristic assumptions inherent in this process are as follows:

1. The best trace starting points for WCET reduction can be identified by looking at the WCET contributions of parts of the WC path.
2. It is sufficient to consider only a limited number (W , L or H) of “most likely” choices at each decision point using the score heuristic.
3. The best starting point for the next trace, e , can be identified by generating traces of length L at each possible starting point.
4. Trace paths should be built in the direction of the greatest WC flow as evaluated before trace building began.
5. Any trace at starting point e can be represented in the IPET model by a surrogate trace with the same starting point e that is chosen from all possible candidate traces to maximise the WCET reduction benefit.

These assumptions only affect the quality of the final allocation, so the timing safety of the resulting allocation is dependent only on the correctness of the IPET model. The correctness of each assumption is reviewed in section 7.3.4.

7.1.12 Phase 2: Selecting Microprograms

The previous phase produces up to HL candidate traces: one for each of the H start points and for each of the L possible lengths. Each is identified by the symbol $T_{e,k}$ in Figure 7.5, with start point e and length k . The space cost of each of these is known ($C_{e,k}$), as is an estimate of the WCET benefit of each ($B_{e,k}$). The problem to be solved in this phase is similar to a 0-1 knapsack problem, because a subset of up to n candidate traces $T_{e,k}$ need to be selected for inclusion in the microprogram store to maximise the total benefit, subject to a cost limit.

However, there is no point in selecting two microprograms with the same start point, so an additional mutual exclusion constraint is added to the problem constraints. If Θ represents the set of chosen microprograms, then this constraint can be expressed formally:

$$T_{e_1,k_1} \in \Theta \wedge T_{e_2,k_2} \in \Theta \implies e_1 \neq e_2 \vee k_1 = k_2 \quad (7.18)$$

Additionally, each WCET benefit estimate $B_{e,k}$ assumes that microprograms have been allocated at all of the preceding start points considered during *Find.Candidates*. The sequence D provides a list of starting points in allocation order, allowing an ordering constraint to be enforced within the second phase. Let D_i be the i -th member of D . Then:

$$\forall j T_{D_i,k_1} \in \Theta \wedge 1 \leq j < i \implies T_{D_j,k_2} \in \Theta \quad (7.19)$$

The effect of these extra constraints is that the selection problem is not a 0-1 knapsack problem. However, an integer linear program can be used to solve it and determine Θ such that WCET is minimised. The program must incorporate constraints representing equations 7.18 and 7.19. In

addition, the cost restriction function is required:

$$\sum_{T_{e,k} \in \Theta} C_{e,k} \leq C_{max} \quad (7.20)$$

And the objective function must be specified:

$$B_T = \sum_{T_{e,k} \in \Theta} B_{e,k} \quad (7.21)$$

The integer linear program must maximise B_T , subject to the space constraint C_{max} . This is done by selecting the contents of Θ , which can be represented in a practical integer linear program solver by *HL* binary variables, one for each possible $T_{e,k}$.

7.1.13 Summary

The algorithms described in this section can be used to model the WCET of a program G including one or more custom microprograms. More generally, these methods can be used to model any trace that conforms to the output of the program described in section 6.3. The algorithms can follow the WC path (Figure 7.4), test various possible reductions, then allocate microprogram store space to obtain an approximation to the optimal WCET reduction (sections 7.1.9 to 7.1.12). Where possible, the algorithms have been derived from previous work, such as the IPET model of Puschner and Schedl, previous methods used to identify worst-case paths [244, 80], and instruction scratchpad allocation algorithms for WCET reduction [244]. Many extensions have been required because of the additional complexity of modelling traces, and the need to select trace path lengths in addition to trace start points during the allocation process.

7.2 Implementing and Testing the WCET Reduction Approach

The algorithms described in the previous section cannot be evaluated fully without a working implementation. The MCGREP-2 CPU generator and its microcode generator provide two of the components required.

The IPET approach for WCET analysis operates from a T-graph, so the first task for an implementation is to obtain the T-graph for the input program (section 7.2.1). Then IPET analysis can be performed (sections 7.2.2 and 7.2.3). It is then possible to carry out the *Find_Candidates* process (section 7.2.4). The final implementation task is microprogram selection (section 7.2.5). Sections 7.2.6 to 7.2.9 describe tests carried out on the software, with a summary in section 7.2.10.

7.2.1 Obtaining the T-graph

Puschner and Schedl [207] state that a T-graph can represent any program, even “unstructured” programs in which (for example) loops and conditional statements are implemented using `goto` operations in place of high-level language constructs. This makes it possible to represent machine code using a T-graph, since machine code has no high-level language features. In fact, a T-graph can be built from machine code containing any of the ORBIS32 instructions (Appendix C), with the single exception of computed jumps which must be handled as a special case.

The implementation described here makes use of an `objdump` listing of the contents of a program binary. It is a standard tool packaged with GNU `binutils`, and it disassembles a binary

T-graph Creation

```

▷ /wctreduce/mergefork.py
Make_Super_CFG: objdump decoder.
Make_BB_S: Merges sequences of instructions to form basic blocks.
▷ /wctreduce/rtas.py
Get_BB_Flows: Converts CFG to T-graph.
Machine_Code_Cost_Model: Obtains machine code execution
costs.

```



program into mnemonic form. (The MCGREP-2 instruction decoder described in section 6.2.1 could also be used for this purpose, but `objdump` is able to incorporate information from the symbol table if one is present, which allows function names to be identified and assists debugging.) This information is used to generate a directed graph in which each node represents an instruction and each edge represents a possible path of control flow.

Basic blocks are formed from this graph by repeatedly merging nodes x and y where x has exactly one successor (y) and y has exactly one predecessor (x). This creates the *control flow graph* (CFG) in which each node is a basic block. (Figure 6.3(a) is an example of a CFG.)

The merging process identifies branch and jump targets in addition to branches and jumps, so it detects all basic block boundaries including join points that are not created by a branch or jump at the end of the previous basic block. However, the process has a weakness: it cannot handle computed jumps (section 6.3.1) as it is not able to identify the destination. One solution is to manually mark each computed jump with a list of possible destinations. Alternatively, the possible destinations could be detected by analysis of the machine code in some special cases, such as `switch` statements. But the easiest solution is to avoid code that generates computed jumps, since the OpenRISC C compiler only produces them for calls to function pointers and complex `switch` statements.

The CFG is converted to a T-graph by two transformations. The first is a conversion to the dual form in which edges and nodes are exchanged. The effect is that basic blocks are represented by edges and branch points are represented by nodes, meeting the T-graph specification. The second is a graph transformation that ensures that every node is connected to no more than three edges, with at most two predecessors and at most two successors. This ensures that there is no need to handle any arrangement of edges within the T-graph as a special case. The transformation involves the creation of new nodes and edges: the new edges are not associated with any machine code and have no effect on functionality.

The T-graph is not complete until each edge $x \in E$ has been labelled with a machine code execution time cost $\gamma(u(x))$. The MCGREP-2 simulator is used to obtain these costs for each edge. The memory latency M is taken from the MCGREP-2 configuration parameter file.

7.2.2 Custom Microcode-Aware WCET Analysis using IPET

This section describes the implementation of a procedure to calculate Z_{G_Θ} , the WCET of G with set of microprograms Θ . For each microprogram $T_e \in \Theta$, it is assumed (1) that every path through

the microprogram $P_{e,i}$ is known, and (2) the execution cost $\gamma(P_{e,i})$ is known for each path.

The procedure carries out two distinct steps. Firstly, it generates a list of constraints and variables to define the integer linear program. Secondly, it solves the integer linear program by maximising Z_{G_Θ} . The second step is examined in section 7.2.3.

The first step uses information from the T-graph. For implementation purposes, the representation of Θ is distributed across two data structures. One of these is within the representation of each T-graph edge. Every edge includes a stack of objects describing the properties of execution of that edge. This is an extension of the context $c(x)$ defined by equation 7.6, and each member of every context includes a WC flow variable f and an execution cost γ . The second data structure is global (with respect to the T-graph): it is a stack containing additional constraints and flow variables, which are combined with the information at each edge to generate the integer linear program. Representing Θ in this way is useful for two reasons:

- It is often necessary to add and remove microprograms from Θ then recompute the WCET. (This is done during every iteration of the main loop in Figure 7.5, for example.) The simplest strategy for doing this, applying then undoing modifications to the T-graph, is undesirable since the code to apply modifications must exactly match the code for undoing them. A computationally expensive strategy would involve copying the entire T-graph before making each change: this approach would be simple to implement, but would not scale well. A better strategy would involve storing the effects of each microprogram separately from the T-graph and combining G and Θ only during analysis. But this is also undesirable because effort would be duplicated (successive evaluations of Θ will usually only differ by one member) and because the microprogram timing model would then exist in two places within the code (trace formation and WCET analysis).

Since custom microprograms are only ever removed in the order they are added, using stacks to represent modifications to the T-graph provides a useful mechanism to allow the effects of custom microprograms to be “rolled back”. The constraints and variables introduced by a microprogram are only placed in the object at the top of each stack, so that their effects can be undone by popping that element. When an operation needs to be finalised (for example, when a surrogate trace is added to Θ), a new object can be pushed onto each stack.

- By associating every flow variable with a specific edge $x \in E$, the WC flow variables representing paths implemented by custom microprograms can be handled by the same mechanism that handles the WC flow variables for machine code. The same is also true of the execution time costs. This simplifies the constraint generation process by removing the need to handle either custom microprograms or machine code as a special case.

Storing the flow through a microprogram at each edge is equivalent to the model described earlier, in which flow through each trace was considered at the path level rather than the edge level. The path level flow variables still exist: they are equivalent to the flow variables at each return to machine code.

The stack mechanism allows constraints to be added during trace formation, and this is used to implement equation 7.11 (trace execution replaces machine code execution at the entrance). It is also used to implement the conservation of flow property at each point where trace execution may split (equation 7.8). The stack mechanism also tracks the new flow variables that are added during

Microprogram Aware IPET Model

```
▷ /wctreduce/rtas.py
```

`Compute_Flows`: Finalises IPET constraints and calls the solver (section 7.2.3).

`ILP_Database`: Container for global constraints and variables.

`Flow_Information`: Container for members of context set $c(x)$ for each edge $x \in E$.



trace formation. The general T-graph constraints are added after the T-graph has been augmented with Θ . These are as follows:

- The developer-provided constraints, which define the bounds on the executions of loops and the existence of infeasible paths (if any). This information is loaded from a text file.
- Execution flow between procedures and functions (the T-graph may be composed of multiple subprograms).
- The defining equation of Z_{G_Θ} (equation 7.15).
- The relationship between $f(x)$ and $f(y)$ where $x \in E$ and $y \in c(x)$ (equation 7.7).
- Conservation of flow for every context at each vertex $v \in V$ (equations 7.9 and 7.12).

The final additions to the integer linear program are ≥ 0 constraints on every variable (since execution flow cannot be negative) and the command to maximise Z_{G_Θ} .

7.2.3 Solving the Integer Linear Program

Integer linear program problems are generally NP-hard, and IPET problems are no exception [159]. This means that $O(2^n)$ steps might be required to compute the maximum Z_{G_Θ} in some cases, although it has been reported that this does not occur if the constraint set is chosen carefully [159].

The solution of integer linear programs is outside the scope of this work, but the subject has already been extensively researched, and solving software already exists. Commercial solvers include CPLEX [132], and free solvers include GLPK [100] and `lp_solve` [166].

The methods used by these programs vary, and it is not clear which of them will be most effective at solving specific problems. Therefore, this implementation makes use of an internal abstraction layer to describe integer linear programs, which allows all of the equations to be specified in a form that is independent of the actual solver. Any solver can be used as a “plug-in” by writing an appropriate backend for this abstraction layer.

Plug-ins have been written for GLPK [100] and `lp_solve` [166]. Both operate by generating the integer linear program equations in a text file, then executing the appropriate solver program. After execution, the results are retrieved by parsing the output (in the case of GLPK) or by calling an API (in the case of `lp_solve`). The values of every flow variable and Z_{G_Θ} are obtained in this way.

Solving Integer Linear Programs

▷ /wctreduce/lp.py
 ILP_Solver: Internal abstraction layer for solving integer linear programming problems.

**Finding Candidates**

▷ /wctreduce/rtas.py
 Generate_Candidates_II: Implements the pseudocode of Figure 7.5.

**Microprogram Selection**

▷ /wctreduce/rtas.py
 ILP_Allocation_Solver: Implements the selection process.

**7.2.4 Finding Candidates**

Candidate traces are identified using the algorithm shown in Figure 7.5. There is little difference between the actual implementation and the abstract algorithm, aside from the use of data structures in place of abstract symbols ($T_{e,k}$, etc.) and the use of the IPET model stack (section 7.2.2) in place of Θ .

However, there is an important extension to the abstract algorithm, making it parameterisable so that the constants H , W and L can be changed and new functions can be used in place of those called from Figure 7.5. For example, the trace formation function can be replaced, along with the score heuristic. These permit experiments to be carried out with alternate versions of these components. The default trace formation function makes use of the microprogram generator described in section 6.3.

7.2.5 Microprogram Selection

The purpose of the microprogram selection step is to choose a subset of the HL trace candidates identified by the algorithm of Figure 7.5. The implementation of this step makes use of the same integer linear program solver used for IPET, via the same abstract interface (section 7.2.3).

Equations 7.18 to 7.21 are translated into integer linear programming constraints by the selection procedure. The solver maximises the total benefit B_T while keeping the total space cost within C_{max} . The results of interest are the values of the decision variables, which indicate the microprograms to be selected. The list of these microprograms is returned as Θ .

Constant	Purpose	Value
H	Maximum number of search iterations.	10
L	Maximum path length to be considered for a trace candidate.	10
W	Number of trace starting points to be tested in each search iteration.	5
C_{max}	Microprogram store space available.	440

Table 7.1: Parameters used for the experiments described in sections 7.2.7 through 7.2.9. The value of C_{max} is the amount of free space in a 512 microinstruction RAM, since 72 lines are allocated to the ORBIS32 interpreter.

7.2.6 Testing the Implementation

The microcode generator and the implementation of TARGET generated by MCGREP-2 have already been tested (section 6.4), but the new components described in the preceding sections need to be verified. This section only considers the functionality of each component, comparing its behaviour to the specifications set out in section 7.1. It does not consider how effective the algorithms described by Figures 7.3 to 7.5 are: this is considered in section 7.3. Because of this, a fixed set of parameter assignments are used for all experiments (Table 7.1).

Three distinct functionality tests have been carried out in order to validate the implementation of the WCET reduction program:

- **Global tests** are applied to a program G_Θ with one or more custom microprograms Θ . The WCET reduction process is applied to the program, which is then tested within the MCGREP-2 simulator. This checks the correctness of the overall solution, including the MCGREP-2 CPU and the trace-style scheduler. Tests of this type are described in section 7.2.7.
- **Local tests** validate the model used to compute $\gamma(T_{e,k})$ for each trace path by checking the exact execution times of the custom microprograms. These are described in section 7.2.8.
- **Explicit path enumeration tests** validate the WCET results for a program with and without custom microprograms. These are described in section 7.2.9.

A new set of benchmarks are introduced for the WCET reduction experiments. Most are from the “WCET analysis benchmark set” distributed by the Mälardalen WCET research group [171]. The specific feature of these benchmarks that makes them useful for WCET reduction experiments is that they are provided with information about loop bounds and infeasible paths, which can be translated into developer constraints for the IPET model. The benchmarks used in earlier chapters (sections 5.3 and 6.4) do not have this feature, so they are not suitable for reuse here.

Not all of the Mälardalen benchmarks are used because of three limitations of the current software. Firstly, programs that use floating point are not considered as MCGREP-2 does not generate hardware to support floating point operations and the overhead of software emulation is significant. Secondly, the WCET analysis subsystem (section 7.2.1) cannot handle computed jumps (section 6.3.1), so some programs that involve `switch` statements are unsuitable. Thirdly, the WCET analysis subsystem does not support recursion.

Name	Description
bs	Binary search
bubble	Bubble sort
cnt	Counts non-negative matrix cells
compress	Compression program
crc	Evaluates a CRC
div	Software division
duff	Copies an array using Duff's Device
edn	Finite Impulse Response filter (1)
expint	Computes exponential integral
fdct	Fast discrete cosine transform (DCT)
fibcall	Fibonacci calculation
fir	Finite Impulse Response filter (2)
insertsort	Insertion sort
janne_complex	Nested loops
jfdctint	DCT on 8x8 pixel block
matmult	Multiplication of 20x20 matrices
ndes	Encryption program
ns	Multi-dimensional array search

Table 7.2: Test programs for WCET reduction. All except `bubble` and `div` are taken from the Mälardalen WCET benchmark set [171].

These issues are specific to hardware generated by MCGREP-2 and the current implementation of the WCET analysis tool. More sophisticated hardware and software designs and better annotation techniques would enable support for floating point, `switch` statements and recursion. However, despite these limitations, many of the Mälardalen benchmarks can be used (Table 7.2).

7.2.7 Global tests

In this section, the WCET reduction process is applied to the benchmarks listed in Table 7.2.

1. **Experiment Goal:** To demonstrate that (a) WCET is reduced by the process described in sections 7.1 and 7.2, (b) the integration of the WCET reduction software and the other MCGREP-2 components functions correctly, and (c) that WCET computations are similar to measured execution times when the Mälardalen developer constraints are used.
2. **Software Setup:** The MCGREP-2 simulator is used within a harness that allows custom microcode to be loaded into the microprogram RAM without needing to recompile each benchmark program. (This extension is important because recompiling the benchmark programs to include a microprogram generator would change the starting addresses of each trace. Since the start points of traces are identified by the WCET reduction process, which uses absolute addresses rather than markers embedded in source code (section 6.4.3), avoiding recompilation is the simplest way to ensure that trace start addresses are unchanged.)

To enable the correctness of the microprogram generator and other components to be tested, the harness also supports checkpoints as described in section 6.4.3.

To allow the execution time of each program to be measured without considering the execution time of initialisation code, the WCET of a single `Benchmark` function is measured in

```

int main ( void )
{
    printf ( "Init\n" ) ;
    fflush ( stdout ) ;
    Initialise () ;
    printf ( "Run\n" ) ;
    fflush ( stdout ) ;
    asm volatile ( "l.nop 0x2001\n" ) ; /* E.T. measure start */
    Benchmark () ;
    asm volatile ( "l.nop 0x2002\n" ) ; /* E.T. measure stop */
    if ( Is_Result_Ok () )
    {
        asm volatile ( "l.nop 0x2005\n" ) ;
        printf ( "Result ok\n" ) ;
    } else {
        asm volatile ( "l.nop 0x2004\n" ) ;
        printf ( "Result NOT ok\n" ) ;
    }
    fflush ( stdout ) ;
    return 0 ;
}

```

Figure 7.6: Framework for measuring the execution time of the `Benchmark` program. The embedded assembly instructions communicate status information to the simulator harness, starting and stopping the timer and signalling success or failure. (A more comprehensive test of correctness is provided by the checkpoint mechanism.)

each case. `Benchmark` replaces the `main()` function of each Mälardalen benchmark. A new implementation (Figure 7.6) provides an alternate `main()` with a test framework to measure the execution time of `Benchmark`.

The memory latency M is set to 1, reusing the assumption that the CPU is connected to large instruction and data scratchpads (section 5.3.6). The array size is 3 units.

3. **Method:** For each benchmark G , the WCET reduction process described in sections 7.1 and 7.2 was applied. This produced a set of custom microprograms Θ , a new program G_Θ , and two WCET values Z_{G_\emptyset} and Z_{G_Θ} . For the purposes of WCET analysis, the program entry point was assumed to be the `Benchmark` function.

The simulator harness generated a “stub” program to load microcode into microprogram RAM. When this completed, the benchmark program was executed. The first execution was carried out without modifying the machine code, so the custom microprograms were unused. This produced E_{G_\emptyset} , the measured execution time for machine code only.

Next, the machine code was modified at each microprogram start point. Measurement was performed again, producing E_{G_Θ} , the measured execution time for custom microcode and machine code together. Checkpoints were also tested during this execution.

The final step compared the four recorded values. Each measurement value must be less than or equal to the corresponding WCET value.

Benchmark G	Machine Code Only WCET Z_{G_0}	M. Code + Microprogram WCET Z_{G_Θ}	WCET Reduction
bs	1.93e+02	1.81e+02	1.066
bubble	2.89e+05	1.73e+05	1.672
cnt	9.62e+04	5.46e+04	1.760
compress	1.25e+04	8.37e+03	1.496
crc	5.91e+04	4.85e+04	1.217
div	6.99e+03	4.86e+03	1.437
duff	5.88e+02	3.20e+02	1.837
edn	1.19e+05	5.96e+04	1.989
expint	1.76e+05	9.05e+04	1.941
fdct	3.68e+03	1.37e+03	2.694
fibcall	7.50e+02	6.88e+02	1.090
fir	1.84e+04	1.23e+04	1.495
insertsort	2.77e+03	1.64e+03	1.685
janne_complex	4.84e+02	4.45e+02	1.088
jfdctint	6.31e+04	3.57e+04	1.766
matmult	9.58e+05	5.34e+05	1.795
ndes	1.47e+05	1.18e+05	1.247
ns	1.59e+04	9.71e+03	1.639

Table 7.3: WCETs for the benchmark programs, before and after the WCET reduction process. Each value is a clock cycle count.

4. **Results:** The values measured for each benchmark are shown in Tables 7.3 and 7.4.
5. **Evaluation:** The results indicate that WCET is reduced by the process. They also show that the various components of the WCET reduction system work correctly together, since no checkpoint tests failed. The “tightness” figures (computed by $\frac{E_{G_\Theta}}{Z_{G_\Theta}}$) indicate that the execution times are close to the WCET estimates for each of the Mälardalen benchmarks. The times are also close for the `div` and `bubble` programs.

The poorest “tightness” values are found for `expint` and `fir`, but these problems are not caused by the WCET reduction process, as they are also present in the results for machine code. The problem is a weak set of constraints for the software division procedure: Mälardalen benchmark constraints assume that constant-time hardware division is used. Simple implementations of IPET assume that procedures are always executed in a single context, making it difficult to accommodate a range of different input values. (Virtual inlining is one solution to this problem [251].)

No WCET underestimates were detected. Unfortunately, these results are not enough to demonstrate that the WCET values are always safe (i.e. $\forall G, E_G \leq Z_G$). Ultimately, the proof that IPET is safe depends on the assumption that the constraints are correct: without this, the true WCET bound cannot be generated.

6. **Observations:** Two interesting properties of traces were discovered.

Firstly, traces mutate the timing properties of a program to the extent that the worst-case input may change. It is no longer safe to assume that worst-case behaviour is produced by the greatest possible number of executions of a piece of code, when the number of executions

Benchmark G	Machine Code Only		M. Code + Microprogram		Tightness
	WCET Z_{G_0}	Measured E_{G_0}	WCET Z_{G_e}	Measured E_{G_e}	
bs	1.93e+02	1.91e+02	1.81e+02	1.77e+02	0.978
bubble	2.89e+05	2.89e+05	1.73e+05	1.72e+05	0.995
cnt	9.62e+04	8.82e+04	5.46e+04	4.46e+04	0.816
compress	1.25e+04	1.21e+04	8.37e+03	8.33e+03	0.995
crc	5.91e+04	5.65e+04	4.85e+04	4.44e+04	0.916
div	6.99e+03	6.46e+03	4.86e+03	4.09e+03	0.841
duff	5.88e+02	5.88e+02	3.20e+02	3.20e+02	1.000
edn	1.19e+05	1.19e+05	5.96e+04	5.96e+04	1.000
expint	1.76e+05	2.13e+04	9.05e+04	1.94e+04	0.214
fdct	3.68e+03	3.68e+03	1.37e+03	1.37e+03	1.000
fibcall	7.50e+02	7.44e+02	6.88e+02	6.29e+02	0.914
fir	1.84e+04	1.37e+04	1.23e+04	6.63e+03	0.538
insertsort	2.77e+03	2.08e+03	1.64e+03	1.27e+03	0.774
janne_complex	4.84e+02	4.66e+02	4.45e+02	4.31e+02	0.969
jfdctint	6.31e+04	5.73e+04	3.57e+04	2.95e+04	0.827
matmult	9.58e+05	8.82e+05	5.34e+05	4.37e+05	0.818
ndes	1.47e+05	1.42e+05	1.18e+05	1.08e+05	0.919
ns	1.59e+04	1.59e+04	9.71e+03	9.14e+03	0.941

Table 7.4: Measured execution times and WCETs for the benchmark programs, providing an estimate of the tightness of the WCET calculation process.

is variable. For example, if a loop may execute up to n times, then the greatest WCET might be produced with $n - 1$ executions, not n , because the effects of loop unrolling may make n executions less costly than $n - 1$. Consequently, it is important to only use equality in developer-specified constraints when the number of executions is fixed. Otherwise, less than or equal time constraints should be used.

Secondly, within the benchmarks considered, traces are allocated in looped code. This is consistent with the usual understanding of hotspots: it is best to apply optimisations in places that are executed more than once. The WCET reduction algorithm has automatically detected those places - the loops. Non-looped code has been avoided not because it cannot be optimised, but because the WCET reductions possible in such areas are less significant. The conclusion must be that programs have to include code that is repeatedly executed in order to gain significant benefits from this WCET reduction approach: there must be *temporal locality*. But this property is also found when caches and scratchpads are used [203], so it is not a specific disadvantage of the use of custom microprograms. Benchmarks without repeatedly executed code can still gain some benefits, since custom microprograms can replace any code.

7. **Conclusion:** WCET reductions are possible using the approach described in sections 7.1 and 7.2. The integration of the WCET reduction software with the MCGREP-2 environment works correctly. Measured execution times are consistent with computed WCETs.

Global Test Framework

▷ /wctreduce/wcetrtest.py
 Test framework control program.

▷ /wctreduce/evaltrace.py
 Measure_Exec_Time: Uses the MCGREP-2 simulator to measure the execution time of a test program G_{Θ} .



7.2.8 Local tests

The correctness of the WCET reduction implementation is predicated on the correctness of the γ values for the edges in the T-graph. These represent either the execution costs of basic blocks $\gamma(u(x))$ or the execution costs of paths through custom microprograms $\gamma(P_{e,k})$.

The execution costs of basic blocks are obtained using the MCGREP-2 simulator. These costs are shown to be independent of execution history and data inputs in section 6.4.7, so using the simulator to measure the execution time of a basic block is sufficient to compute its $\gamma(u(x))$ value. However, measuring the execution costs of paths through custom microprograms is more challenging, because many different data-dependent paths are possible. Merely capturing the address of the exit point is not sufficient to identify the path, because (1) multiple paths can lead to the same exit, and (2) some traces contain repeat operations (section 6.3.4), so a very large number of paths may exist.

1. **Experiment Goal:** To demonstrate that the measured execution times of paths through custom microprograms exactly match the execution times expected by the timing model.
2. **Software Setup:** It is necessary to capture information about each path taken through a custom microprogram in addition to the total execution time. This is implemented by an extension to the MCGREP-2 simulator.

The function of this extension is as follows. Whenever custom microcode begins executing, each taken branch is recorded using the `HOOK_UPC_BRANCH` simulator hook (Table 6.5). When the custom microcode ends, causing a return to machine code, the branch information is passed to a validating function which is generated by the trace timing model, along with the exit address and the measured execution time. The validating function checks this input and aborts the experiment if there is a mismatch.

The software setup is otherwise identical to the description in section 7.2.7.

3. **Method:** The experiment of section 7.2.7 was repeated with the additional extension described above. Tests were also performed with different memory latency values: M from 1 to 5.
4. **Results:** The results confirmed that there is no discrepancy between the timing model and measured execution times.
5. **Conclusion:** The timing model for custom microcode is correct.

Local Test Framework

```
▷ /wctreduce/wcetrtest.py
```

Test framework control program.

```
▷ /wctreduce/rtas.py
```

Apply_Trace: Generates Validator function.

**7.2.9 Explicit Path Enumeration Tests**

It is not generally possible to validate WCET figures by measurement of a single-path through a program, but it is nevertheless important to ensure that the WCET values produced are correct. A comparison with another WCET calculation approach would be useful. Unfortunately, the Z_{G_0} WCET values are not comparable with those obtained using other WCET analysis tools since no other user of the Mälardalen benchmarks has analysed programs with the ORBIS32 ISA.

“Explicit path enumeration” (section 2.1.1) involves the measurement of the execution time of every possible path through a program. This is generally intractable, but it is possible for specific cases. One algorithm with well-defined worst-case behaviour is “bubble sort”, which has $O(n^2)$ time complexity when used to sort a list of n items. Since only the relative values of the input list are relevant, there are $n!$ possible paths through a bubble sort routine: one for each possible permutation of the input.

For sufficiently small n , it is possible to explicitly measure the execution time of each of these permutations. The maximum time is the “measured WCET”. If this is less than the computed WCET (from the extended IPET model), then the constraint set is incomplete. If this is greater than the computed WCET, then one or more of the constraint set, IPET model and implementation are incorrect.

1. **Experiment Goal:** To compare two ways of calculating the WCET of the bubble sort algorithm (as implemented in Figure 7.7). Explicit measurement of every possible path is carried out using the simulator to obtain the true WCET, and the extended IPET model is used to calculate the expected WCET.
2. **Software Setup:** The code in Figure 7.7 is used (the `bubble` benchmark for a list of $n = 7$ elements), using some developer-specified constraints (Figure 7.8).

There are $7! = 5040$ input permutations to test. The test harness is an extension to the MCGREP-2 simulator that provides the functions required by the experimental method. The harness generates each input permutation in turn, then measures its execution time.

3. **Method:** For each permutation, the following operations were carried out:
 - The permutation was loaded into the bubble sort program by modifying the binary.
 - The `Benchmark` function was executed, with the execution time being measured by the framework in Figure 7.6.
 - The execution time was recorded.

```
#define SORT_SIZE 7
void Benchmark ( void ) {
    int    i , swapped ;
    do {
        swapped = 0 ;
        for ( i = 0 ; i < ( SORT_SIZE - 1 ) ; i ++ ) {
            int    si0 = to_sort [ i ] ;
            int    si1 = to_sort [ i + 1 ] ;
            if ( si0 > si1 ) { /* swap */
                to_sort [ i ] = si1 ;           /* BB4 */
                to_sort [ i + 1 ] = si0 ;
                swapped = 1 ;
            }
            /* BB5 */
        }
        /* BB6 */
    } while ( swapped ) ;
}
```

Figure 7.7: Bubble sort C source.

BB4 <= 21 ;
BB5 <= 42 ;
BB6 <= 7 ;

Figure 7.8: Developer-specified constraints for Figure 7.7. The constraints apply to the basic blocks marked with `/* BBn */`.

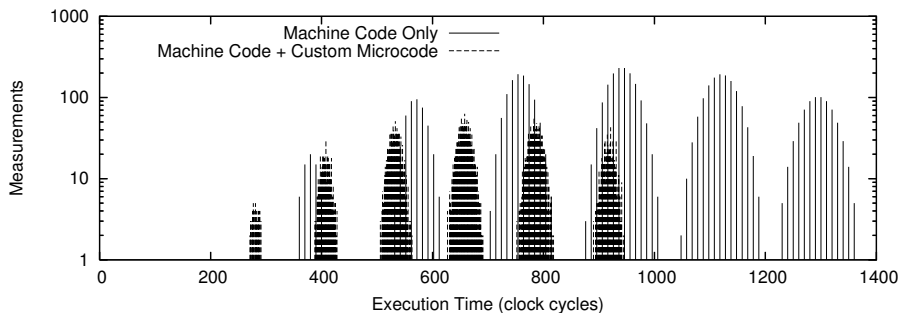


Figure 7.9: Bar chart of measured execution times obtained from the bubble sort algorithm when executed with each of the 5040 possible input permutations.

	$x = \emptyset$	$x = \Theta$
Z_{G_x}	1370	1007
Z'_{G_x}	1370	951
W.C. Input	(6, 5, 4, 3, 2, 1, 0)	(6, 4, 3, 2, 1, 5, 0)

Table 7.5: Maximum execution times Z' and computed WCETs Z for the the bubble sort algorithm.

These operations were firstly carried out using machine code only in order to obtain Z'_{G_\emptyset} : the maximum execution time obtained by explicit path enumeration for machine code execution.

Then the WCET reduction process was applied to find Θ , Z_{G_\emptyset} and Z_{G_Θ} . The operations were repeated to find Z'_{G_Θ} , the maximum execution time obtained by explicit path enumeration for machine code execution combined with custom microcode.

- Results:** All of the execution times measured in this experiment are shown in Figure 7.9 and the WC path times are shown in Table 7.5.
- Evaluation:** These results confirm that the WCET computations for the bubble sort program meet the criteria established in section 2.1. They are safe, in that they do not underestimate the actual maximum execution time (i.e. $Z'_{G_x} \leq Z_{G_x}$). They are also tight, in that each computed WCET is close to the maximum execution time.

However, the input permutation that results in the maximum execution time is changed by the introduction of custom microcode. For machine code, the worst-case behaviour of this bubble sort implementation is produced by inputs in reverse order. But this is not true when a custom microprogram is in use (Table 7.5). This is because the trace that is generated for the bubble sort algorithm unrolls the loop. The unrolling includes the assumption that a `/* swap */` in one iteration (Figure 7.7) will be followed by further swap operations in subsequent iterations. Worst-case behaviour is no longer produced when many swap operations are required in adjacent iterations. Now, it is produced when iterations requiring a swap are separated by iterations that do not require a swap. This demonstrates the trace property described in the evaluation of section 7.2.7: developer-specified constraints should set upper bounds rather than exact values when the worst-case number of executions is not known. Reducing the effects of one WC path just introduces another: and this new WC path might be activated by different input conditions. However, analysis is no more difficult in this scenario, because an

Explicit Path Enumeration

```
▷ /wctreduce/experiment2.py
```

Runs explicit path enumeration experiment.



IPET model can be created to model the effects of the traces and determine Z_{G_Θ} .

6. **Conclusion:** The results indicate that the IPET model works correctly for bubble sort, and also demonstrate that reducing the execution time of the WC path can change the input conditions required to produce the maximum execution time.

7.2.10 Summary

The results of sections 7.2.7 through 7.2.9 indicate that the WCET reduction program works correctly in conjunction with the MCGREP-2 implementation described in chapter 6. WCET reductions have been obtained for each of the programs while preserving functional correctness. The experiments also indicate that the timing model and the IPET process work correctly. It is observed that WCET reductions tend to target loops in preference to other code (section 7.2.7.6), indicating that temporal locality is important to maximise benefits from WCET reduction.

7.3 Effectiveness of the WCET Reduction Algorithm

The results from the previous section show that the WCET reduction algorithm works. However, those results do not provide any information about how close the algorithm comes to the maximum possible WCET reduction for a specific program. Generally, it is not possible to say how close a particular WCET reduction approach comes to optimal, because of the influence of the architecture and the program on the WCET reductions that can be applied, but it is possible to look at the improvements that could make an algorithm more effective. This section begins with an examination of the parameters of the WCET reduction algorithm and their effects (sections 7.3.1 to 7.3.3). This is followed by an examination of the assumptions made by the algorithm in section 7.3.4.

The WCET reduction algorithm has four parameters. Three of these affect the first phase of the algorithm, described by pseudocode in Figures 7.3 to 7.5. The fourth (C_{max}) affects the second phase, in which the microprogram store space is allocated. These parameters were given fixed values for the tests in sections 7.2.7 through 7.2.9 (Table 7.1). But these fixed values were essentially arbitrary choices: other selections might produce improved results, or work better with certain types of program. The purpose of each parameter is as follows:

- L is the maximum path length to be considered for any trace candidate. Length is measured by the number of branch points. Longer traces may permit greater WCET reductions, although as section 6.4.11 noted, increasing the length of a trace does not necessarily reduce either ACET or WCET.

- H is the maximum number of search iterations to be executed during the first phase of the search process. Each search iteration produces L candidate traces (for the second phase) and one surrogate trace (so that the WC path is changed for the next iteration).
- W is the number of trace starting points to be tested during each search iteration. The starting point that leads to the greatest WCET reduction is chosen for the candidate and surrogate traces.
- C_{max} is the total space available within the microprogram store.

One of the effects of increasing L , H and W is that searches will take longer. Similarly, decreasing these parameters makes searches complete in less time. More trace candidates will need to be generated if L is increased, more iterations will be attempted if H is increased, and more start points will be evaluated if W is increased. But more subtle effects of these parameters also apply. These are described in sections 7.3.1 to 7.3.3.

7.3.1 Maximum Path Length L , and Microprogram Store Space C_{max}

The maximum path length L is best considered in conjunction with the microprogram store space, C_{max} . Increasing the size of a trace is worthless without also increasing the space that the trace can expand into. This is confirmed by Figure 7.10, where various values of L and C_{max} have been used on each benchmark. (The method described in section 7.2.7 was reused.)

These results show that increasing L can allow greater WCET reductions to be obtained, but the magnitude of the possible improvement is limited by C_{max} . However, it is also clear that relatively small values of L such as 4 or 5 are sufficient for most of the benchmarks. Large values of L do not lead to substantial extra benefits, but very small values of L should also be avoided.

This is related to the effects observed in section 6.4.11: increasing the length of a trace does not necessarily improve its performance. (The WCET reduction algorithm naturally ignores suboptimal candidates by evaluating the exact WCET benefit of each.) Providing additional space for traces *can* improve performance, though, because it allows *more* traces to be used. This effect is independent of L , but can be observed in the change in WCET reduction as C_{max} is increased. For example, the WCET of the `fdct` benchmark is not affected by L , but is nevertheless reduced as C_{max} increases. This is because additional WC paths are optimised using the extra space.

7.3.2 Maximum Number of Search Iterations, H

H controls the number of search iterations that are attempted during the execution of the algorithm in Figure 7.5. Each iteration finds a set of candidate traces and a single surrogate trace, all with the same start point.

Because candidate traces have to be selected for inclusion in the microprogram store *in the order they were generated*, increasing the value of H has only one effect: more candidates are considered. This only reduces the WCET further if there is space in the microprogram store for additional traces. If there is not, then the search effort is wasted.

Therefore, H provides a simple way to reduce the search costs for any particular program. It can be set to the number of traces selected by previous executions, plus a small margin to allow new traces to be determined, and only increased if the number of selections increases. This is helpful in an iterative development process.

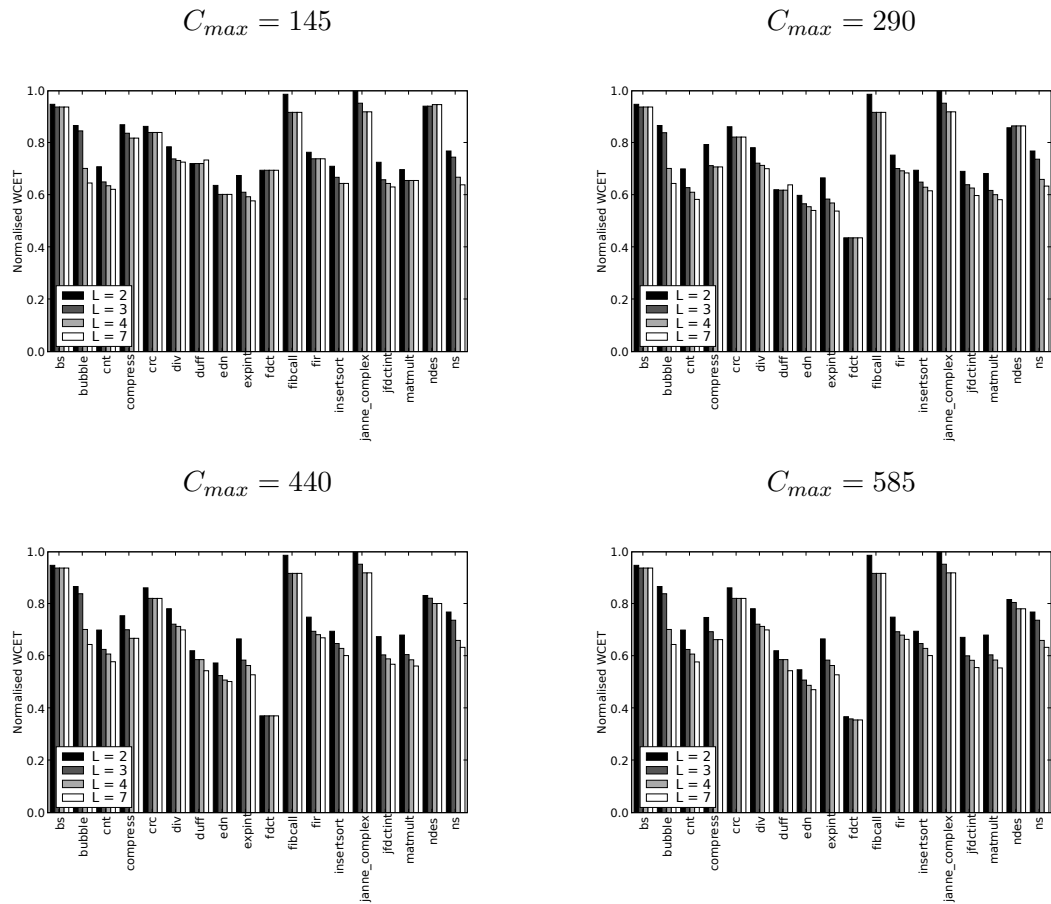


Figure 7.10: Effects of changing the L and C_{max} parameters on benchmark WCETs, normalised against machine code WCETs.

Changing L and C_{max}

▷ `/wctreduce/varyparams.py`

Experimental tool that tests different values of L and C_{max} .



The ordering restriction is described by equation 7.19, and is needed because the WCET reductions produced by trace candidates are not independent from each other. The WCET reduction provided by each trace may be dependent on one or more of the WCET reductions produced by traces already in Θ .

7.3.3 Number of Trace Start Points to Test, W

W is the number of trace starting points to be tested during each search iteration. This is a form of “search window”. The W greatest contributors to WCET are located using the score heuristic (Figure 7.3). Then, the actual WCET reductions created by traces starting at those points are evaluated.

The intuition for this process is that the score heuristic is not perfect. Ideally, the heuristic would indicate exactly what WCET reduction would be possible given a particular start point. But there is no way to calculate this without generating a trace and computing the WCET. Therefore, an estimate is used instead.

This estimate does not indicate exactly how well the WCET reduction process will work, but it does indicate the magnitude of improvement that is possible under idealised conditions. It only considers the total execution time and WC flow through the subsequent basic blocks, which does not provide any information about the results of generating a trace. In reality, the effectiveness of a trace depends on available ILP, the bias of the branches, the number of memory accesses, and the data dependences in the code. (This is clearly demonstrated by the variability of the execution time reductions for benchmark programs, e.g. Figure 6.19.)

Increasing W leads to better quality results because each of the W start positions identified by the heuristic is evaluated by trace generation and a WCET computation. This is a realistic evaluation that shows exactly what WCET reduction is possible from that start point. Provided that the score heuristic can always place the best candidate within the top W results, the algorithm will work correctly. However, increasing W also increases search time and leads to diminishing returns: if the score heuristic always places the best candidate in the top x results, then there is no point in testing $W > x$ because $W - x$ tests will always be discarded.

This expectation is matched by the experimental results shown in Figure 7.11. For some benchmarks, changes to W do not affect the WCET attained at all. In these cases, the best candidate is identified with $W = 1$: the score heuristic is working perfectly. For some other benchmarks, such as `cruc`, there is a slight variation in WCET as W is increased: this happens when multiple choices are available within the search and increasing W changes the set of choices being considered, so the order of allocation may change. `bubble` and `ndes` are examples of cases in which the score heuristic has failed: for `ndes`, each increase in W results in a drop in WCET, indicating that the score heuristic misidentified the best starting point. Better solutions are identified as the window size is increased.

The cause is a general problem with identifying the best start point within looped code, which affects the operation of the score heuristic but not the other parts of the WCET reduction algorithm. Consider a loop containing one conditional statement, like the inner loop of `bubble` (Figure 7.7). In the absence of developer-specified constraints, IPET will assume that the statement is always executed because this maximises Z . Therefore, the WC flow around the conditional statement will be zero, and the flow through it will be non-zero. In this situation, the score heuristic will not be able to distinguish between at least three possible starting points: (1) the loop entry block, (2) the

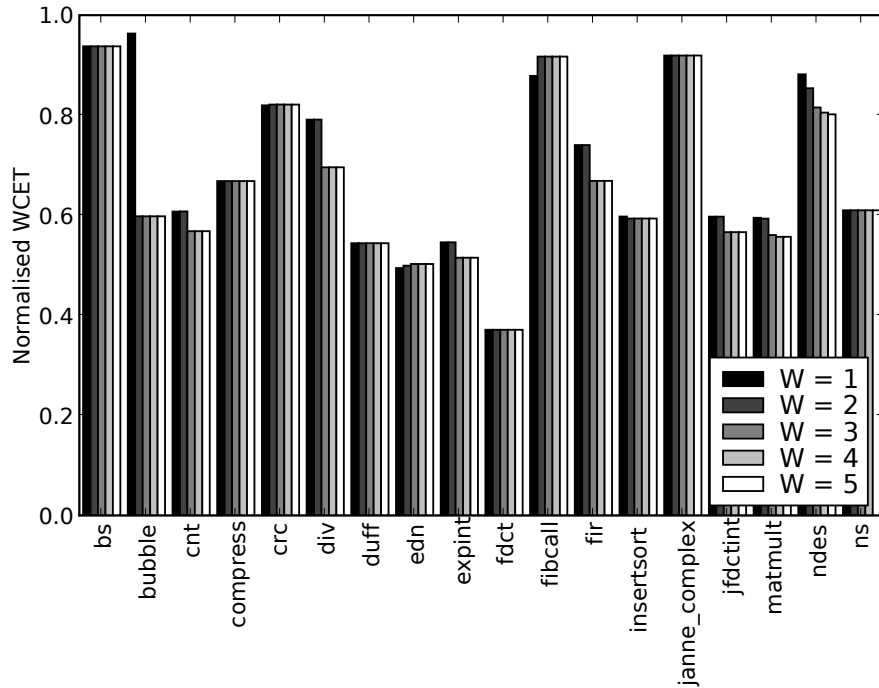


Figure 7.11: Effects of changing W on the normalised WCET of each benchmark. Other parameters set as shown in Table 7.1, and each result is normalised against the original (machine code only) WCET.

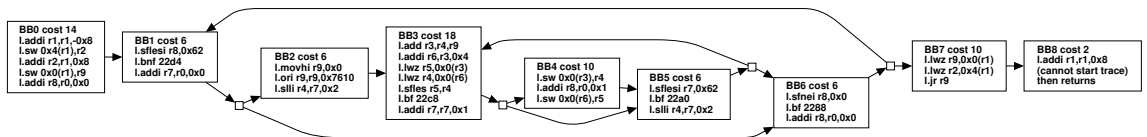


Figure 7.12: Basic block graph of bubble (Figure 7.7).

conditional statement, and (3) the loop repeat block. All three are part of the same loop, and all three have the same WC flow value.

Examples of these three starting points are shown for `bubble` in Figure 7.12, where the entry block is BB3, the conditional statement is BB4, and the repeat block is BB5. In this case, BB5 is the best start point, because executions of BB5 lead to executions of BB3 and BB4 (in the worst case). But this is only apparent to the score heuristic if the developer specifies that the conditional statement BB4 is not always executed. (This is done by the constraints listed in Figure 7.8.)

In the absence of such constraints, the same WCET reductions can still be obtained by increasing W to the number of basic blocks along the WC path within the loop. This is clearly not ideal. It would be useful to handle situations in which internal constraints are missing without forcing W to be increased, since increasing W adds substantially to the execution costs of the search. This is considered in section 7.3.6.

7.3.4 Assumptions

The WCET reduction algorithm includes some implicit assumptions, which are stated in section 7.1.11. This section challenges each one.

1. **The best trace starting points for WCET reduction can be identified by looking at the WCET contributions of parts of the WC path.**

This assumption is based on previous work, which locates the best places to apply optimisations by using information about the contributions of each part of the program to the total WCET. Suhendra [244], Puaut [202] and Falk [80] all make use of this approach. Zhao [295] also uses WC path information to determine the best targets for optimisation.

Most usefully, Suhendra [244] evaluated a “greedy heuristic” based on repeatedly selecting the greatest WCET contributors against an exhaustive search. Suhendra found that the heuristic approach produced similar results to exhaustive search, while being scalable to larger programs. Puaut’s results [202, 203] confirm this finding. This is good evidence for the correctness of the assumption.

2. **It is sufficient to consider only a limited number (W , L or H) of “most likely” choices at each decision point using the score heuristic.**

While reducing the search space to the most likely candidates is a good strategy, it includes the implicit assumption that the mechanism used to select these candidates works correctly. This mechanism is implemented by the score heuristic (Figure 7.3) and the WC path finding heuristic (Figure 7.4). Section 7.3.3 gives an example of a scenario in which the score heuristic fails to identify the best candidate: clearly, the assumption is not perfect.

However, the algorithm does reduce WCET, so perhaps the assumption is “good enough”. A quantitative evaluation is required. One way to do this is to increase W , L and H to infinity. In the case of L , this is impossible because of limitations in the MCGREP-2 software. But section 7.3.1 has already discussed the effects of increasing L , which cease to provide any benefit after a certain threshold, so a relatively small number (e.g. $L = 10$) will suffice. In the cases of W and H , this can be simulated by using the maximum number of basic blocks in any test program. The search process is still bounded by the size of the benchmark program, but since $HW + HL + 1$ IPET computations are required, the search time is substantially increased.

A simple experiment was performed to compare the use of large values of W and H (50 and 1000, respectively) with those obtained using the parameter settings shown in Table 7.1. The WCET reductions obtained in each case were almost identical (Table 7.6, columns 2 and 3). This evidence indicates that the score heuristic works well enough for the benchmarks considered, and as the benchmarks are intended to be representative of general software, this suggests that the heuristic will be effectively for most (if not all) programs.

3. **The best starting point for the next trace, e , can be identified by generating traces of length L at each possible starting point.**

The intuition behind this assumption is that the longest possible trace will provide the same (or better) WCET reduction than shorter traces with the same starting point. This saves testing L possible traces at each starting point, which would change the number of IPET

Benchmark	$H = 10$	$H = 1000$	$H = 10$
	$W = 5$	$W = 50$	$H = 5$
	1 tested	1 tested	L tested
bs	1.81e+02	1.81e+02	1.81e+02
bubble	1.73e+05	1.73e+05	1.73e+05
cnt	5.46e+04	5.46e+04	5.46e+04
compress	8.37e+03	8.37e+03	8.37e+03
crc	4.85e+04	4.85e+04	4.85e+04
div	4.86e+03	4.86e+03	4.86e+03
duff	3.20e+02	3.20e+02	3.20e+02
edn	5.96e+04	5.96e+04	5.96e+04
expint	9.05e+04	9.05e+04	9.05e+04
fdct	1.37e+03	1.37e+03	1.37e+03
fibcall	6.88e+02	6.88e+02	6.88e+02
fir	1.23e+04	1.23e+04	1.23e+04
insertsort	1.64e+03	1.64e+03	1.64e+03
janne_complex	4.45e+02	4.45e+02	4.45e+02
jfdctint	3.57e+04	3.57e+04	3.57e+04
matmult	5.34e+05	5.34e+05	5.34e+05
ndes	1.18e+05	1.18e+05	1.18e+05
ns	9.71e+03	9.71e+03	9.71e+03

Table 7.6: Evaluating assumptions by removing them. Column 2 shows the reduced WCET of each program from Table 7.3. Columns 3 and 4 show the reduced WCETs minus assumptions 2 and 3 respectively.

computations required from $HW + HL + 1$ to $HWL + 1$. However, long traces may not provide the greatest WCET reduction benefit. Consider the WCET reduction values listed in Table 7.7, showing the relationship between trace length and WCET reduction for the `bubble` benchmark. The maximum value of L , 8, does not provide the largest WCET reduction. Therefore, a suboptimal starting point might be identified.

This assumption can be quantitatively tested by examining the effects of removing it. This is done by testing each starting point with L different trace lengths rather than one. A simple modification to the algorithm shown in Figure 7.5 makes this possible. Experiments indicate that the change makes almost no difference to the eventual WCET reduction (Table 7.6, columns 2 and 4).

4. Trace paths should be built in the direction of the greatest WC flow as evaluated before trace building began.

This assumption does not necessarily hold. WC flow values are changed by the addition of a trace, so it is possible that an earlier part of a trace could change flow to the extent that later parts of the trace would take a different direction if WCET analysis were performed during trace path building.

However, results from section 7.3.1 suggest that the effects of extending a trace are easier to characterise. As the trace length increases, the benefits of further extensions diminish. This suggests that the assumption is incorrect only in the sense that WC flow is reduced as the trace gets longer. Repeating WCET analysis during trace path building will not reveal new

L	WCET Reduction
1	830900
2	661800
3	990100
4	905900
5	1044016
6	979075
7	1070950
8	1013620

Table 7.7: WCET reduction caused by traces of length L beginning at basic block BB5 of `bubble` (Figure 7.12). The longest trace does not provide the greatest WCET reduction. This is due to the two branches within the `bubble` inner loop. The greatest WCET reduction is obtained when the final trace exit matches up with the trace entrance.

Testing The Assumptions

▷ `/wctreduce/varyparams.py`

Experimental tool for testing different configuration settings and alternate heuristics.

▷ `/wctreduce/compare_results.py`

Result comparison tool.



paths, because the longer the trace is, the less benefit is available.

5. Any trace at starting point e can be represented in the IPET model by a surrogate trace with the same starting point e that is chosen from all possible candidate traces to maximise the WCET reduction benefit.

This is a “best-case scenario” assumption. Any of the candidate traces could be selected for start point e by the second phase of the algorithm. But *one* of them must represent them all so that further WC paths can be identified. So it is assumed that the second phase will be able to select the trace resulting in the greatest WCET reduction. This does not capture the possible effects of different trace lengths on subsequent selections.

Unfortunately, it is not clear how to avoid this assumption. Like assumption 1, this assumption is a consequence of the search strategy itself. The starting locations for WCET reductions after the first can only be found if earlier WCET reductions are already applied. Selecting a single surrogate trace makes this possible.

7.3.5 Improvements: Search Strategy

The previous section indicates that some of the assumptions inherent in the WCET reduction algorithm are not perfect. Where possible, experiments have indicated that the algorithm nevertheless works well in practice. But improvements can still be made, and this section discusses some possibilities.

Assumptions 1 and 5 are effectively a part of the search strategy. Removing these assumptions would mean applying a different type of search entirely. When there are many possibilities for optimising a program, but the search space is complex due to the possible interactions between choices, heuristic search methods such as simulated annealing have often been applied. For example, simulated annealing has been applied to the classical co-design problem [79] which is related to this WCET reduction problem in that the available optimisations are subject to a resource limit and a selective process is needed to allocate resources to the greatest effect.

Simulated annealing [146] would make it possible to iteratively improve any WCET reduction solution. The search would need to determine Θ : a collection of trace starting points with trace lengths. The aim of the search would be to minimise the WCET as evaluated using IPET. This would be achieved by making random changes to Θ and then reevaluating Z_{G_Θ} , keeping the changes if an improvement has been made, and discarding the changes if the space limitation is exceeded. The magnitude of the changes would be gradually reduced, with the result that Θ would be likely to reach a good (if not optimal) solution for each program.

A similar strategy could be applied using other heuristic search methods such as genetic algorithms and tabu search. Any of these heuristics could reuse existing components of the WCET reduction algorithm, as all that is needed in principle is a way to evaluate a candidate Θ using Z_{G_Θ} and the total space cost.

However, such heuristic searches will reach the goal only as a result of random operation. Such an “evolutionary” approach is known to be effective in search problems of extreme complexity, but the approach has the disadvantage of requiring very long computation times. The searches cannot apply knowledge about how the goal of a minimal Z_{G_Θ} should be reached, and consequently the improvements are the result of chance rather than intelligence. Thus, execution time is unbounded. This is not suitable for the requirements as they include the need for an efficient optimisation process.

The WCET reduction problem is framed in such a way that intelligence can be applied: (1) the resource consumption of each trace can be expressed by an integer, (2) the greatest contributors to WCET can be targeted first, and (3) WCET reductions can be applied iteratively. The score heuristic guides the WCET reduction algorithm towards the places where traces will have the greatest effect, by estimating the reduction available from a trace in those positions. And a knapsack-like problem can be solved to allocate resources efficiently.

This allows the complete execution time of the algorithm to be a matter of minutes for even the most complex benchmarks considered. This sort of strategy could not be applied to classical co-design because resource consumption in that problem has many more dimensions than a single integer. But it can be applied here, cutting the search time to a bounded number of IPET computations.

Because of the assumptions within the existing algorithm, simulated annealing and genetic algorithms might produce better results. However, there would be no clear bound on the number of iterations required to reach a good solution, and the solutions obtained might vary between searches. Exploring the value of these heuristics is a topic for future work.

7.3.6 Improvements: Score Heuristic

The score heuristic does not work perfectly (section 7.3.3). It is defeated by conditional statements in the absence of developer-specified constraints. IPET-based WCET analysis always assumes that

the most costly execution path is taken when constraints are not provided, and this can make the basic blocks within a loop indistinguishable to the heuristic, which only considers the WC flow (Figure 7.3).

The best way to avoid this problem is to add new constraints to specify the conditional statement behaviour within loops. However, it is worth considering whether enhancements could be made to the heuristic to avoid or reduce the problem. A variety of possible enhancements have been considered:

1. **Explicitly targeting a specific point** within a loop, e.g. the repeat block, by zeroing the scores of all other blocks.

This approach does not work well in general because the best start point in one loop is not the best start point in another. For example, the best trace start point in the `div` benchmark is a conditional statement, whereas the best trace start point in the `bubble` benchmark is the exit block (Appendix F).

2. **Attempting to incorporate changes in flow** within the WC path identified by the score heuristic.

The first score heuristic (Figure 7.3) assumes that the flow is constant throughout each trace. This is incorrect if any side exit is ever taken. To improve the accuracy of the model, the cost of each path can be scaled by the best-case fraction of the flow at the trace start point. The changes to the heuristic are as follows:

- Initial scale factor $s_e = 1$.
- On every transition from path edge x_i to path edge x_{i+1} , if $f(x_i) > f(x_{i+1})$, then $s_{x_{i+1}} = s_{x_i} \frac{f(x_{i+1})}{f(x_i)}$. Otherwise, $s_{x_{i+1}} = s_{x_i}$.
- $e.\text{dom_cost} = \sum_{x \in P_e} \gamma(x) s_{x_i}$

Although this model is arguably more realistic, it makes no difference to the quality of the results obtained. Unfortunately, it is still unable to select the best starting point in some loops.

3. **Explicitly detecting unconstrained conditional statements within loops.**

There is a way to identify unconstrained conditional statements in loops. Each is characterised by two acyclic paths between two vertices. After IPET analysis, one path has zero WC flow, and the other has non-zero WC flow. A heuristic can detect subgraphs with this property, and then make a new assumption:

Unconstrained conditional statements within loops are equally likely to be executed or not executed.

This assumption could not be applied within WCET analysis, since it is incorrect in general. However, for the purposes of the score heuristic, the assumption is useful because it causes starting point selections to avoid conditional code. Any unconstrained split point (vertex with two exits) is assumed to divide flow in half, which is represented using a scale factor: $s_{x_{i+1}} = \frac{1}{2} s_{x_i}$. The same thing happens at an unconstrained join point (vertex with two entrances) because only half of the flow into that vertex will be part of a trace.

There are two problems with this enhancement. Firstly, there is another situation in which a path might have zero WC flow: a block that is not on any feasible path has zero WC flow

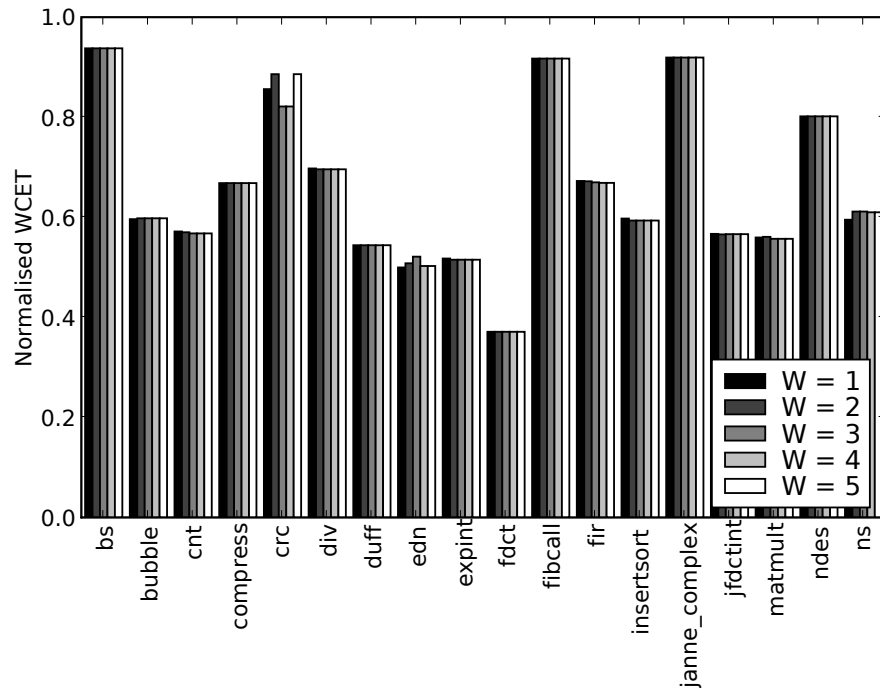


Figure 7.13: Effects of changing W on the normalised WCET of each benchmark, using an alternate version of the score heuristic.

by definition (section 2.1.2). In such a case, the developer-specified constraints are complete, but the enhanced score heuristic will misidentify such unconstrained branches. Secondly, the two paths may not be equally likely. In this case, the conditional statement may actually be the best trace start point.

The effects of enhancements 2 and 3 are shown in Figure 7.13, where the effects of changing W are reduced in comparison to Figure 7.11. However, the performance of the new score heuristic is poorer for the `div` and `edn` benchmarks. This is because the assumption of enhancement 3 has failed. The unconstrained conditional branches within the `div` benchmark are executed more often than not.

Ultimately, accurate developer-specified constraints are needed to represent program behaviour for WCET analysis purposes. Without such constraints, the results of analysis may not be tight (section 2.1). When the WCET is being reduced by an algorithm such as the one described here, a lack of constraints can reduce the effectiveness of the WCET reduction process. To some extent, this can be addressed by heuristics, but these cannot be perfect.

7.4 Implementing Scratchpad Allocation

In chapters 5, 6 and 7, it has been assumed that large instruction and data scratchpads can be used to reduce memory access latency to a single clock cycle (sections 5.3.6, 6.4.8 and 7.2.7). This assumption is not scalable, because the size of any scratchpad is limited by chip area. In practice,

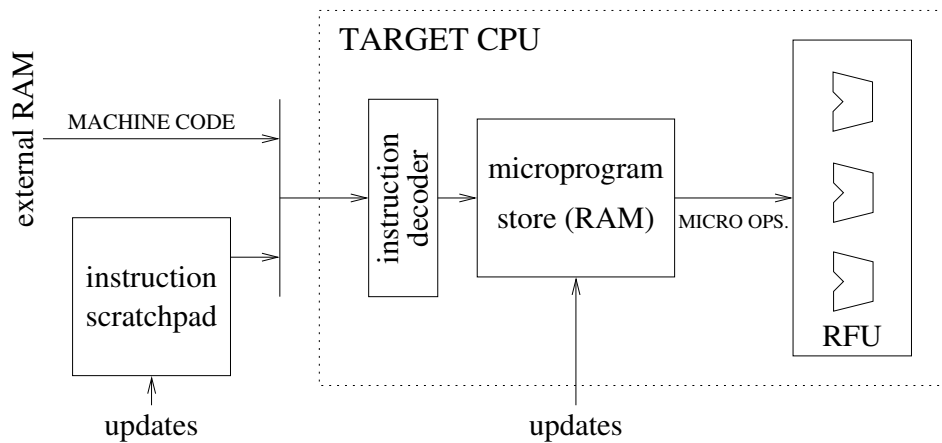


Figure 7.14: Instruction scratchpad extension for TARGET. Instructions may come from one of three places: instruction scratchpad, external RAM, or the microprogram store. The scratchpad and microprogram store are both small low-latency memories. The external RAM is large, but each access requires a number of clock cycles.

scratchpads only hold a small part of the entire program memory at once. The purpose of scratchpad allocation algorithms such as [240, 241, 139, 244, 202, 273] is to make best use of this space (that is, minimise ACET, energy consumption or WCET).

To fully evaluate TARGET, it is important (1) to remove the “large scratchpad” assumption, and (2) to compare the WCET reduction algorithm described in section 7.1 against previous work, including scratchpad allocation algorithms. This creates two additional requirements: an instruction scratchpad extension for TARGET (Figure 7.14) and a WCET reduction algorithm for scratchpads (section 7.4.1).

7.4.1 Implementing Suhendra’s Instruction Scratchpad Allocation Algorithm

A small set of changes can be applied to the microprogram store allocation algorithm (Figure 7.5) in order to make it equivalent to Suhendra’s algorithm (Figure 2.9). It is possible to consider each basic block within an instruction scratchpad as a form of trace, each with a single-path that replaces a single edge of the T-graph.

The microprogram store allocation procedures (section 7.2) are designed to be extensible with new features. Extensions can be used to substitute the trace model for an instruction scratchpad model, just by changing the “apply” function that adds a trace to the extended IPET model to support a new type of “trace”, and setting $L = 1$ and $W = 1$. (Each “trace” is a basic block with a fixed length $L = 1$, and Suhendra’s algorithm considers a single candidate per iteration, so $W = 1$.) With these changes, the second phase is redundant with the implementation described in section 7.1.12, because only one allocation decision is possible at each point. (There is only one possible length for each “trace”.) Additionally, $H \geq |E|$ for instruction scratchpad allocation so that all basic blocks may be considered.

The resulting implementation provides a way to improve upon Suhendra’s algorithm since W possible “worst-case” candidates can be tested by IPET analysis at each decision point. $f(x)\gamma(x)$ is not a perfect way to evaluate the possible WCET reduction from allocating x , because the actual WCET reduction depends on the other parts of the T-graph. Therefore, $W = 5$ is used for both

Instruction Scratchpad Reduction Algorithm

```
▷ /wctreduce/rtas.py
```

Apply_InstSP: adds a “trace” representing the movement of a basic block into an instruction scratchpad.



microprogram store and instruction scratchpad allocation, since this will improve upon the results that would be obtained with an exact implementation of Suhendra’s algorithm [244].

Following the implementation of Suhendra’s algorithm, the abstract WCET reduction process can be extended to accommodate instruction scratchpad allocation (Figure 7.15). With this extension, the process can support WCET reductions using an instruction scratchpad, a microprogram store, or both. This enables comparison with previous work, and allows the “large scratchpad” assumption of sections 5.3.6, 6.4.8 and 7.2.7 to be avoided in later experiments.

7.5 Summary

This chapter has demonstrated the first part of the hypothesis (section 3.4): that WCET reductions are possible by exploiting ILP in run-time reconfigurable hardware. It has also shown that TARGET can meet all of the requirements set out in section 3.2 with the appropriate WCET reduction software. The WCET reduction algorithm makes some assumptions, but all of these have been validated using evidence from experiments or previous work. Improvements and alternative search strategies have also been considered. Finally, extensions have been implemented to support instruction scratchpad allocations.

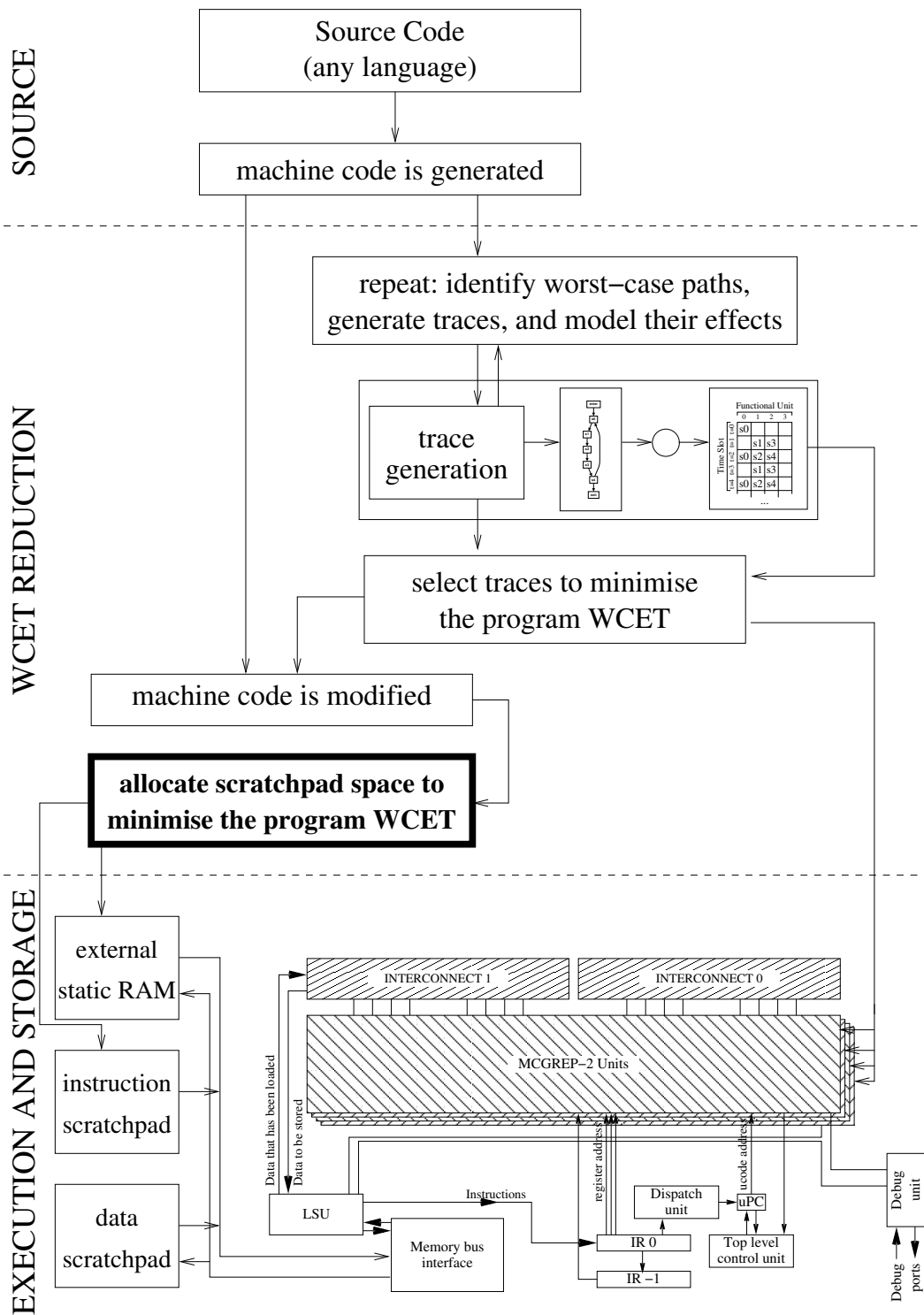


Figure 7.15: The instruction scratchpad allocation algorithm for WCET reduction (section 7.4.1) fits into the abstract WCET reduction process as illustrated here: the new component is highlighted in bold.

Chapter 8

Evaluation

In this chapter, the three evaluation criteria from section 3.5 are examined:

- Is the hypothesis (section 3.4) demonstrated?
- Is an implementation of the CPU architecture functionally correct?
- Does an implementation satisfy the requirements listed in section 3.2?

The second item has already been satisfied by the tests carried out in sections 5.3, 6.4 and 7.2.7, which tested the correctness of each of the parts of the abstract WCET reduction process (Figure 4.1). The third item is also satisfied by the TARGET implementation generated by MCGREP-2, as section 6.4.14 describes.

However, the hypothesis is not fully demonstrated. The implementation of the IPET-based WCET reduction approach in chapter 7 completes the implementation of the abstract WCET reduction process and demonstrates the first part of the hypothesis, since program WCETs can be reduced. But this does not demonstrate the second part of the hypothesis, specifically that WCET can be reduced further than possible using comparable previous work.

Therefore, other ways to reduce WCET must be examined. From chapter 2, these can be classified into the following groups:

- Using separate RAM for instructions and data (a Harvard architecture);
- Merging common sequences of instructions into single instructions implemented by non-reconfigurable CPU hardware (the ASIP and CISC approach);
- Using a conventional cache to reduce the effects of the memory bottleneck [195];
- Using a locked cache or scratchpad to reduce the effects of the memory bottleneck [80, 202, 244, 273];
- Using a dynamic superscalar issue unit to exploit ILP in code [236];
- Using VLIW instructions to exploit ILP in code [87];
- Exploiting ILP within a single basic block [217];
- Using a multi-core or SMT architecture to execute several threads at once (section 2.2.2.6);
- Offloading tasks to co-processors [20, 239, 1].

Not all of these approaches meet the requirements stated in section 3.2:

- Those involving conventional caches, either directly (when a cache is introduced to reduce the memory bottleneck) or indirectly (when the memory bottleneck is increased due to multi-core, SMT, superscalar or VLIW operation), do not meet requirement 2: the basic block timing invariance property. This is because the time taken to execute the code depends on the execution history, as this affects the contents of the cache.
- Those involving co-processors may lose the scalability requirement (number 5) because the number of co-processors is bounded by hardware limitations. For the same reason, the ASIP/CISC approach of merging common instructions together does not scale. Run-time reconfiguration (section 2.6.5) can be used to introduce hardware virtualisation on an FPGA, but this does not help within a CPU because of the limitations of FPGA RTR.
- Those involving co-processors are also limited by the fact that a single language cannot efficiently describe both software and hardware [106], so they do not meet requirement 1: support for general software programs. The use of place and route means that requirement 4 is not met, as the optimisation process will be slow.
- Using a Harvard architecture does not solve the memory bottleneck problem as it can only double the available memory bandwidth. So it does not meet the scalability requirement.

This leaves just two approaches that meet the requirements. These are the use of a locked cache or scratchpad to reduce the memory bottleneck [80, 202, 244, 273], and the exploitation of ILP within a single basic block [217]. The WCET reduction research carried out using these approaches is directly comparable to TARGET and the work described in chapter 7.

Sections 8.1 and 8.2 compare TARGET with these two alternative approaches. Experiments are described in sections 8.3 and 8.4. The results are evaluated together in section 8.5.

8.1 Exploiting ILP Within a Single Basic Block

Evidence from previous work [186, 268] indicates that ILP is limited to approximately two parallel instructions when parallelism is restricted to a single basic block. This problem motivated the introduction of global compaction for microprograms (section 2.6.8). Efficient global compaction was enabled by Fisher's trace scheduler [85], which was later used for VLIW code generation [87] where it allowed the number of instructions executed in parallel to be increased towards the theoretical limit of parallelism within the code [268].

Clearly, local compaction (ILP within one basic block) is undesirable, but when basic block execution times can change, WCET analysis is more difficult. If it is not possible to exploit ILP across basic block boundaries, then local compaction is better than no compaction at all. The limitations of the VISA "simple mode" [9], which permits in order execution only, are reduced by exploiting ILP within a single basic block [217].

However, as chapter 7 has demonstrated, it is possible to use global compaction via traces and still apply WCET analysis using methods that depend on basic block timing invariance. Because global compaction is known to permit greater parallelism than local compaction or in order execution [186, 268, 87], it is evident that TARGET's approach will allow greater WCET reductions in principle.

8.2 Comparison with Instruction Scratchpads

The operation of the WCET reduction algorithm described in section 7.1 is similar to an instruction scratchpad allocation algorithm in that code is being allocated to faster memory. In section 7.4, extensions to the algorithm were made in order to implement Suhendra’s instruction scratchpad allocation algorithm. This enables direct comparison between the new approach and its predecessor: instruction scratchpad allocation.

A microprogram store can allow greater exploitation of ILP for the reasons explored in section 8.1, but this is at the cost of increased complexity in the software used to allocate the space and translate the software. Some of the allocation algorithm steps are simply unnecessary for instruction scratchpad allocation. For example, there is no need to build or evaluate traces of different lengths, since basic blocks are copied into scratchpad without changes. The effects of migrating a basic block x to instruction scratchpad are easy to model by a change in execution cost $\gamma(x)$ within an IPET analysis model. And the hardware itself is simpler: a custom microprogrammable CPU architecture like TARGET is not required, because the scratchpad can simply be added to an existing CPU (Figure 7.14). Even this step may be unnecessary if a cache locking feature can be used to simulate a scratchpad [80].

This section asks whether the additional complexity of TARGET is worthwhile in practice. This is done by comparing scratchpad architectures with custom microprogrammed architectures. The experiments make use of the benchmark set from chapter 7 (Table 7.2) and use the architecture shown in Figure 7.14. This allows three different configurations to be compared:

- **Configuration 1: Instruction Scratchpad**

The CPU only executes machine code, which comes from external RAM or the instruction scratchpad. The microprogram store is not used.

- **Configuration 2: Custom Microprograms**

The CPU executes machine code from external RAM, and custom microcode from the microprogram store. The instruction scratchpad is not used.

- **Configuration 3: Hybrid**

The CPU executes machine code or custom microcode. Machine code may be executed from external RAM or the instruction scratchpad. Custom microcode is executed from the microprogram store.

All three configurations meet the requirements (section 3.2) in that they can all be used to reduce the WCET of programs while retaining scalability, ease of WCET analysis, and support for any program. In each case, the effects of the memory bottleneck can be reduced, so WCET is not necessarily bounded by the speed of external RAM accesses.

8.3 Comparison Experiment

1. **Experiment Goal:** To evaluate the WCET reduction possibilities for benchmark programs in the three different scenarios identified in section 8.2:

- (a) Instruction Scratchpad,

Memory Type	Latency	Length	Units for Length	Size in Bits
Inst. Scratchpad	1	512	instructions	16384
Inst. External RAM	10	infinite		
Microprogram Store	N/A	143	lines	16302
Data Scratchpad	1	infinite		

Table 8.1: Architectural configuration for the scenario comparison expression. The latency of the microprogram store is not applicable (N/A) because of integration into the CPU pipeline.

- (b) Custom Microcode,
- (c) Hybrid (both of the above).

In each scenario, it is assumed that a data scratchpad algorithm such as the one described by Suhendra [244] is used to minimise the latency of every data access. This means that the data access latency is always 1, whereas the instruction fetch latency is dependent on whether the instruction is in the scratchpad (latency 1) or not (latency determined by external RAM).

The experiment also evaluates the relative merits of allocating memory to each type of memory by setting the scratchpad size (in bits) to match the microprogram store size (in bits).

2. **Software Setup:** A WCET allocation algorithm for instruction scratchpads is used (section 7.4.1).

Extensions for the MCGREP-2 simulator can support multiple RAM latencies, as shown in Table 8.1. The instruction scratchpad is implemented as a transparent overlay on external RAM, effectively acting as a locked instruction cache. (In practice, scratchpad memories are normally placed in an alternate address space, then reached by branch instructions, since this saves decoding logic at the cost of forcing the allocation process to make code modifications. A transparent overlay gives equivalent behaviour, requires no code modifications and is easier to implement within a simulator.)

The external RAM latency is assumed to be 10 clock cycles in order to represent a likely RAM latency for an embedded system. (This is the approximate latency for cache misses in the Microblaze uClinux system discussed in section D.6.)

3. **Method:** The experimental framework described in section 7.2.7 was extended to test WCET reductions in each scenario.

An upper bound was selected for the scratchpad memory space, set to 16384 bits for this experiment. Setting the microprogram store size to 143 lines gets close to this setting, as 114 bits are required for each line on an 2 unit CPU generated by MCGREP-2 and configured for the ORBIS32 ISA. This allows the total sizes of the two types of memory to be closely matched by choosing the appropriate architectural parameters (Table 8.1).

The algorithmic parameters shown in Table 8.2 were used by the WCET reduction process. For the combined configurations, instruction scratchpad space was allocated first: this ordering simplifies the implementation of the tool.

4. **Results:** The results of this experiment are shown in Figure 8.1 and Table 8.3.

Constant	Purpose	Value
H	Maximum number of search iterations for microprogram store allocation.	10
H'	Maximum number of search iterations for instruction scratchpad allocation.	1000
L	Maximum path length to be considered for a trace candidate during microprogram store allocation.	10
L'	Maximum path length considered during instruction scratchpad allocation.	1
W	Number of starting points to be tested in each search iteration.	5
C_{max}	Microprogram store space available.	143

Table 8.2: Parameters used for the experiment described in section 8.3.

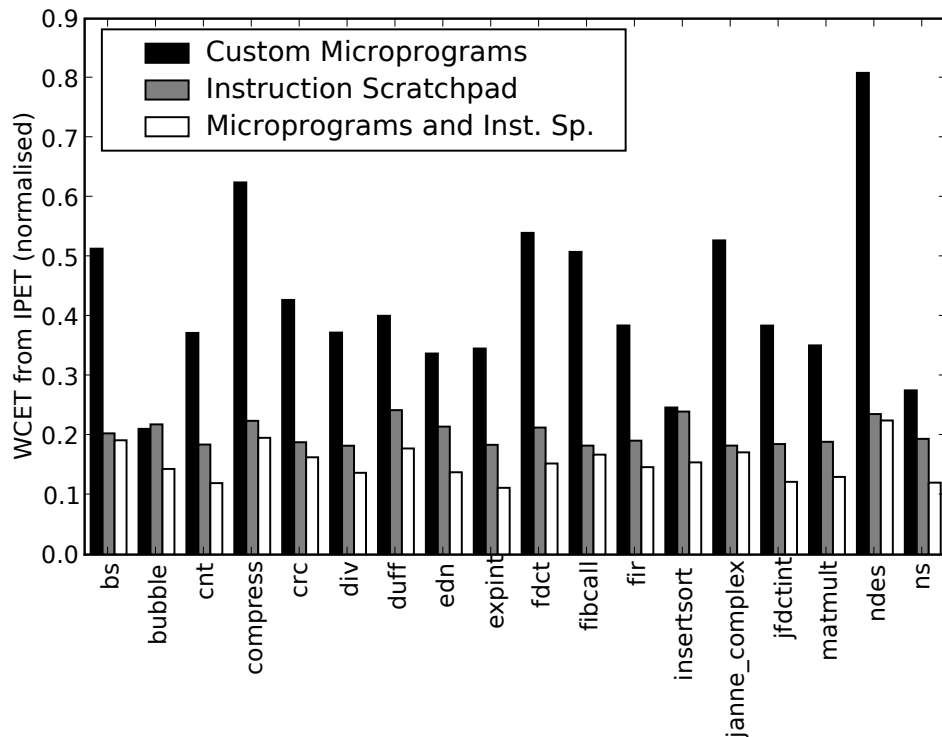


Figure 8.1: Comparison of the WCET reduction capabilities of instruction scratchpads, custom microcode, and hybrid architectures combining both.

Program	Microprogram Bits Used	WC Exec. in RFU (%)	Inst Sp. Bits Used	WC Exec. in Inst. Sp. (%)	W.C. Reduction (%)
bs	8322	19.2	1472	80.8	6.0
bubble	12426	89.0	992	11.0	52.0
cnt	15960	55.2	5952	44.8	53.9
compress	15504	40.7	16320	29.5	14.5
crc	14706	55.5	5376	44.5	15.4
div	15960	64.1	4192	35.9	33.1
duff	14364	63.9	1568	36.1	36.1
edn	15390	60.7	16160	39.3	55.4
expint	15960	56.2	6400	43.8	64.4
fdct	15732	31.4	7968	68.6	39.5
fibcall	9234	72.7	768	27.3	9.0
fir	16188	55.7	6048	44.3	30.3
insertsort	15960	79.0	2016	21.0	55.1
janne_complex	10830	39.2	1152	60.8	6.6
jfdctint	15960	53.2	10272	46.8	52.0
matmult	16302	33.8	6976	66.2	45.2
ndes	13338	16.1	16096	72.2	4.8
ns	12996	52.8	1792	47.2	60.9

Table 8.3: Detailed information about the results of Figure 8.1, showing how space is allocated in the hybrid configuration.

5. **Evaluation:** The results illustrate that custom microcode can improve upon the WCET reductions possible using instruction scratchpads in some cases (e.g. `fdct`). Instruction scratchpads compare well to custom microcode when used in isolation because machine code is far more *dense* than microinstructions in terms of operations per memory bit, so *more* of the program can be present in the scratchpad.

More importantly, however, the results indicate that combining custom microprograms and an instruction scratchpad within a single system always leads to the best WCET reduction: better than either technology alone. Therefore, the abstract WCET reduction process should include both technologies (Figure 8.2).

The experiment can be extended to examine other allocations of memory space. None of the benchmark programs were large enough to fill the instruction scratchpad (Table 8.3), so a smaller instruction scratchpad could be used. The experimental method was repeated using different memory sizes from 1 kilobyte (8192 bits) to 8 kilobytes (65536 bits) with different MCGREP-2 array sizes. For each benchmark program and each configuration, the total WCET reduction between a hybrid configuration and an instruction scratchpad configuration was computed (Table B.5) and plotted as a graph (Figure 8.3).

These results illustrate that WCET reductions are available using small MCGREP-2 configurations. Although allocating 65536 bits of memory to both the instruction scratchpad and the microprogram store does lead to the greatest WCET reductions in the examples tested (e.g. `fdct`), smaller arrangements also provide good results. However, the results also expose a limitation of the trace generator, which is unable to generate a trace for part of a basic block. This becomes a problem when microprogram store space is very limited and the WC path

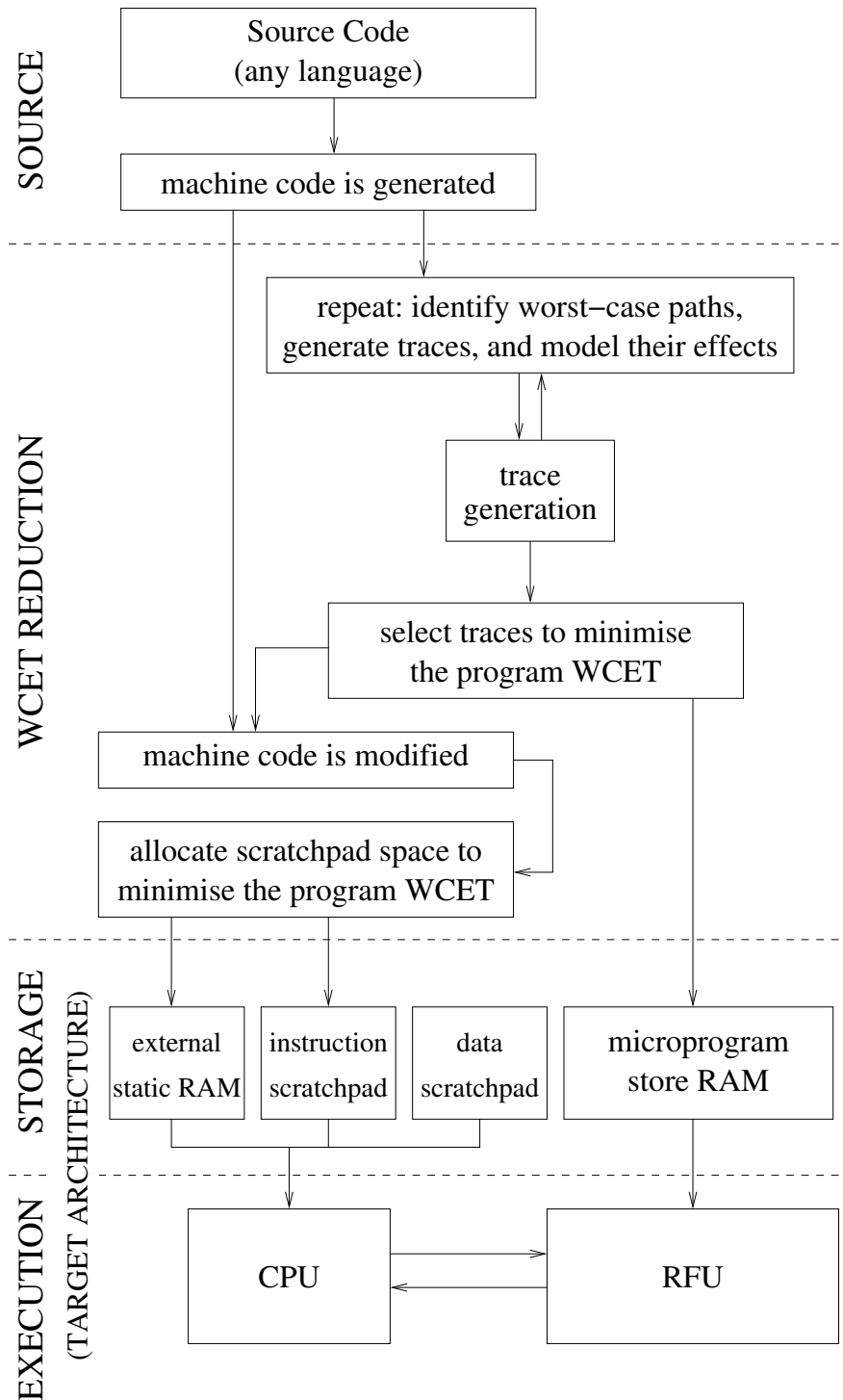


Figure 8.2: The results indicate that the abstract WCET reduction process should incorporate both instruction scratchpad allocation and custom microprograms in order to minimise the WCET of a program. This is reflected by this updated version of Figure 4.6.

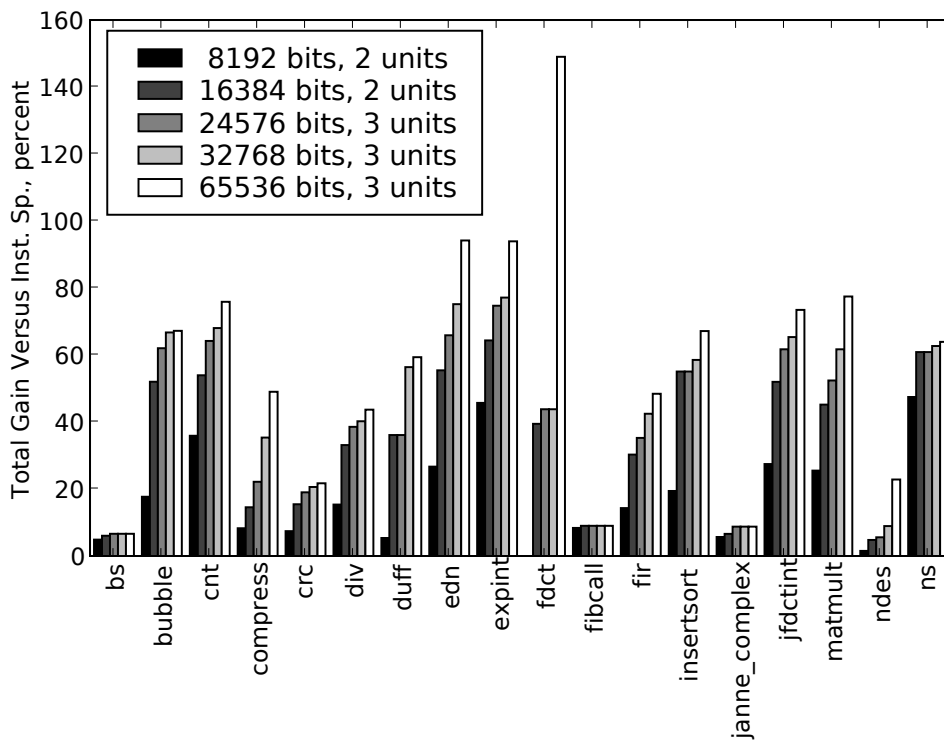


Figure 8.3: Improvements made by a hybrid (scratchpad and RFU) architecture versus an instruction scratchpad architecture. Each result is a percentage representing the WCET reduction for each program and configuration. This data also appears in Table B.5.

includes very long basic blocks, as in `fdct`. A minimum of 16384 bits of microprogram store space were needed for the `fdct` WC path.

- Conclusion:** Custom microcode can reduce the WCET of some programs to a greater extent than instruction scratchpads. However, the two technologies should be used together to achieve the best results, since they have different strengths. Custom microprograms are good for the most utilised code requiring the greatest WCET reductions, while instruction scratchpads are good for larger quantities of less commonly used code.

8.4 Additional Experiments

The work described in section 8.3 is extended in Appendix G, where support for Puaut’s scratchpad allocation algorithm [202] is implemented, and some of the benchmarks from Table 7.2 are partitioned into multiple regions. It is known that TARGET can support large programs by partitioning (sections 4.3.4, 5.3.7 and 6.4.10), so the WCET reduction algorithm in chapter 7 could always be applied to any task by solving each region separately. Puaut’s algorithm integrates an awareness of loading time into this process, which is a subtle improvement. The results of Appendix G confirm the results of section 8.3: microprogram store allocation permits WCET reductions beyond those possible using an instruction scratchpad only. This is true even when the program is partitioned and

loading time is taken into account.

8.5 Summary

The experiments in this chapter have demonstrated that an instance of TARGET can provide greater WCET reductions than previous approaches matching all other requirements (section 3.2).

The best WCET reductions seen in experiments using MCGREP-2 CPUs are achieved by combining instruction scratchpads and custom microcode. This allows programs to execute from faster scratchpad memory, reducing the effects of the memory bottleneck. It also allows some parts of a program to execute from microprogram memory, which reduces the instruction rate bottleneck. Therefore, hybrid solutions (as seen in Figure 7.14) are suggested as a predictable implementation strategy for embedded real-time systems. These provide the best way to reduce the WCET of a program while retaining amenability to timing analysis (as defined in section 3.1).

Therefore, instances of TARGET are a solution to the problem of reducing the WCET of a program without introducing WCET analysis issues. CPUs generated by MCGREP-2 are not well optimised, but they can reduce program WCETs substantially (Figure 8.3). It is likely that future instances of TARGET could provide even greater WCET reductions.

Chapter 9

Conclusion

This work has investigated the possibility of using a predictable CPU architecture to reduce the worst-case execution time of tasks in a hard real-time system. The hypothesis (section 3.4) stated that:

Run-time reconfigurable (RTR) hardware can be used to exploit instruction level parallelism (ILP) within software in order to reduce the worst case execution time (WCET) of a program without adding significant complexity to the WCET analysis process. The application of this technique will reduce the worst-case effects of the instruction rate bottleneck (identified earlier) to a greater extent than previous work in the area of predictable CPU architectures, these being CPU designs that are specifically intended to support WCET analysis. The technique will also be demonstrated to scale to support large programs.

Each part of this hypothesis has been demonstrated. The first part of the hypothesis is demonstrated by the results in sections 6.4.8, 7.2 and 8.3. Experiments show that MCGREP-2 CPUs exploit ILP in software to reduce the WCET. These results apply more generally to all CPUs based on the TARGET architecture class (chapter 4).

The technique has been compared with existing architectures in the same class (section 8.3), and the results indicate that the approach improves upon previous work, as required by the second part of the hypothesis. Then, experiments with partitioning demonstrate that the TARGET-based WCET reduction approach described in section 7.1 can scale to programs of any size (sections 4.3.4, 5.3.7, 6.4.10 and Appendix G) as required by the third part of the hypothesis. Therefore, instances of the TARGET architecture can be used to reduce the WCET of software without introducing analysis difficulties.

Section 9.1 summarises the findings of the work. Section 9.2 suggests possible future developments. Finally, section 9.3 concludes.

9.1 Summary of Findings

To reduce the WCET of a general program while preserving amenability to WCET analysis, execution time reductions need to target worst-case paths, which may be composed of any machine code (section 3.1). In general, good co-processor implementations cannot be built from software languages [106], so speedups need to be applied within the processor itself. This suggests an RFU

implementation within a configurable CPU, with automatic microprogram generation by a trace-style acyclic scheduler (section 6.1.4).

Experimental results (section 8.3) indicate that custom microcode can improve upon instruction scratchpads [244, 202] in most cases. Combining the two technologies helps: instruction scratchpads and custom microprograms always reduce the WCET of a program over either technology alone.

Because of their different properties, instruction scratchpads and custom microcode are a good fit for two types of code within a program. The basic blocks that make the most significant contribution to the WCET can be placed within the microprogram store, because that provides the greatest WCET reduction at the highest memory cost. Less frequently used code fits well within the instruction scratchpad: this does not provide such a good WCET reduction, but the memory cost is very low (32 bits per instruction). The least used code is found in external RAM. This is the cheapest memory, and it is also the slowest: the access bottleneck severely reduces code speed.

Therefore, the conclusion must be that the two technologies should be used together. This result echoes research from microprogramming and dynamic caching. The paradigm of splitting microprogram stores into two distinct units, with one containing narrower instructions than the other, has been widely explored. It is used within the Motorola 68000 [257] and earlier microprogrammed architectures such as the Nanodata QM-1 [218]. The QM-1 allowed both the *horizontal* (wide) and *vertical* (narrow) stores to be reprogrammed. The vertical store is analogous to the instruction scratchpad in Figure 7.14, and the horizontal store is analogous to the microprogram store generated by MCGREP-2.

More recently, dynamic caches have also been split with separate stores for microinstructions and machine instructions. A trace cache [221, 220] (section 2.2.1) stores the result of decoding machine instructions *in their most likely order of execution*. This allows pipelining to be improved, since accesses to the cache usually happen in sequential order. A trace cache is always combined with an instruction cache in a practical architecture. Together, they act as transparently reprogrammed vertical and horizontal microprogram stores, with the cache update mechanisms supervising the application of automatic hardware-controlled optimisations. Instruction caches and trace caches have both been used in practical architectures [135].

As predictable cousins of instruction caches and trace caches respectively, instruction scratchpads [244, 202] and writable control stores [218] can provide WCET reduction benefits while retaining compatibility with IPET analysis (section 7.1.8, [244]). Provided that the CPU control components and memory architecture are predictable, the combination of the two allows the WCET of a program to be reduced further than either component alone (section 8.3). The TARGET architectural class encompasses the required CPU features (section 3.2).

The WCET analysis research field is faced with many problems caused by ACET optimisations in current CPU architectures. These include the timing anomaly problem (section 2.3.7) and the complex modelling issues for recent CPUs [121]. Current WCET analysis technology can model moderately complex CPUs statically [80], and probabilistic techniques have also proved to be a successful way to generate models by measurement [34].

However, a step towards architectures that simplify analysis has also been advocated as an alternative [204, 9, 69, 273, 80]. Solutions have already been implemented [244, 80, 217], but these only allow limited ILP to be exploited during predictable execution. The solution proposed in this work, TARGET, allows more ILP to be found using execution traces which are compatible with an extended form of IPET (section 7.1).

Thus, the MCGREP-2 CPU generator for the TARGET architecture (chapter 6) combined with the results in chapter 8 demonstrate that statically programmed WCET reductions are: (1) not limited to sequential execution, or ILP within a single basic block; and (2) amenable to WCET analysis by IPET. In fact, the degree of WCET reduction can be increased by adding memory to the microprogram store and the instruction scratchpad, while retaining scalability to any program size through partitioning (Appendix G). The degree of improvement can also be increased by adding functional units to the CPU (section 6.4.11).

9.2 Future Work

There are four primary areas for future work based on this thesis. These are as follows:

- **Answering Further Questions about Extended IPET:**

The IPET approach (section 2.1.2) can be easily applied to any program on a CPU with basic block timing invariance. The problem is only slightly more difficult if microcode is also used (section 7.1): each only adds an extremely limited set of extra paths.

Adding more possible paths will eventually cause the IPET problem to become too complex to solve in a reasonable timeframe [280, 183]. This did not occur for any of the benchmarks considered, suggesting that the number of extra paths used within the benchmark programs is well below this complexity horizon, which is unlikely to be well-defined in any case.

However, this leads to two questions: (1) how many traces can be added to a program for TARGET before the extended IPET problem becomes intractable? And (2), what extensions to the general form of a trace (as modelled in section 7.1.3) can be incorporated into an IPET model?

This work has considered only a small number of traces, because traces are limited by the microprogram store size. It has also only considered a single trace model. Large numbers of traces could be used if memory requirements were reduced. Alternative ways to schedule code such as modulo scheduling (section 6.1.3) could provide greater WCET reductions. These features would still meet the requirements for TARGET if they could be modelled by IPET extensions while retaining scalability.

- **Improved TARGET Derivatives:**

MCGREP-2 CPUs (chapter 6) are not as optimised as they could be. The microprogram interpreter uses at least two clock cycles for every instruction (Table 6.8). This design was introduced because the operation of the CPU is bounded by memory speed (section 4.2.1), but when an instruction scratchpad is used, this bottleneck is removed. Therefore, a more advanced design is needed in which one instruction can be executed per clock cycle. Classical RISC designs could form a basis for this [195].

According to section 6.4.13, the microcode execution subsystem is also not optimised. Some writeback operations may add an unnecessary overhead to trace execution. Additionally, branches within traces make use of ALU functionality, so there is a reduction in the effective array size during branch operations. Both of these problems could be reduced by a more sophisticated design. Zero-cost (parallel) branches have been implemented in other types of

CPU, and writebacks could be reduced or eliminated entirely by moving register renaming operations into hardware.

The RFU interconnect should also be optimised to reduce hardware usage and increase the maximum possible clock frequency. This could be done by using a clustered architecture [87] to implement the RFU, and by pipelining transfers between functional units. The trace generation algorithm would need to be improved to support this.

Generally, future generations of TARGET derivatives could apply many of the optimisations used in VLIW CPUs (section 6.4.13). The only restriction on features is posed by the requirement for simple WCET analysis, which forces components to operate in a predictable way. Currently, this means executing standard machine code, or executing a trace that has been considered as part of WCET analysis. However, extensions to the IPET model might permit other execution modes to allow greater WCET reduction.

As an extensible implementation generator of a subset of TARGET with accompanying tools, MCGREP-2 provides a good basis for these improvements. The instruction set and microinstruction set can be reconfigured and extended using functions written in Python, VHDL and C.

- **Alternative Approaches for TARGET:**

The implementation of TARGET described in chapter 4 makes use of a reconfigurable functional unit (RFU) to implement parts of a WC path. In effect, WCET is reduced by migrating WC path fragments into an alternate form of CPU which can execute them more quickly. The technique is limited by the microprogram store memory that is available.

As an alternative, the RFU could be operated by the output of a dynamic superscalar issue unit rather than traces stored in microprogram memory, *provided* that the operation of the dynamic unit is constrained to fit the expectations of an earlier analysis phase. This would mean that it could not carry out data-dependent operations such as dynamic memory disambiguation, and that worst-case branch directions would have to be encoded in the program in some way. A clearly defined border would be enforced between accelerated code and in-order execution to allow the analysis approach of chapter 7 to be reused.

The benefit of this approach is that the microoperations do not need to be stored, so an instruction scratchpad is sufficient to allow them to be reconstructed in the form expected by analysis. This would allow more traces to be used within a program, potentially allowing the WCET to be reduced further. It might also be easier to adapt an existing CPU design into this form. However, energy consumption and hardware requirements might be increased.

This is somewhat similar to VISA [9], where the operation of a complex CPU is constrained to a simple timing model. The difference is that all of the dynamic operations of the CPU are predicted during analysis, so the minimum WCET is not constrained by a simple model. It is also related to WCET analysis approaches applied to complex CPUs, such as [121], but differs from these in that only one set of dynamic operations is possible in each situation.

- **Improved WCET Reduction Tools:**

The WCET reduction tools could be improved in at least two ways. Firstly, partitioning can enable better use of scratchpad space (Appendix G), but partitioning is currently only supported by a simple heuristic mechanism. This could be improved, potentially improving

the WCET reduction that can be achieved for some programs. Secondly, an interface for interpreting WCET constraint annotations in standard formats would improve the ease of use of the WCET analysis tool (section 7.2).

9.3 Conclusion

This thesis has used experiments on working hardware models and simulated equivalents to demonstrate a way to extend simple IPET approaches to support instruction-level parallelism. In so doing, it extends previous work involving predictable CPUs and predictable memory architectures. The work has characterised the architecture for a type of predictable CPU to be supported by extensions to IPET analysis. These extensions have then been specified and implemented in a WCET reduction program. A VHDL implementation of the predictable CPU has been used to demonstrate these features on an embedded system. Experimental results indicate that the hypothesis has been satisfied.

Chapter 10

References

- [1] J. Agron, W. Peck, E. Anderson, D. Andrews, E. Komp, R. Sass, F. Baijot, and J. Stevens. Run-Time Services for Hybrid CPU/FPGA Systems on Chip. In *Proc. RTSS*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.
- [2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
- [3] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proc. POPL*, pages 177–189, New York, NY, USA, 1983. ACM Press.
- [4] P. Altenbernd. On the false path problem in hard real-time programs. *ECRTS*, 00:0102, 1996.
- [5] Altera. Hardcopy II Structured ASICs: ASIC gain without the pain (accessed 28 September 07). <http://www.altera.com/products/devices/hardcopyii/hr2-index.jsp>.
- [6] Altera. Classic EPLD Family. Data sheet A-DS-CLASSIC-05, 2005.
- [7] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks. Architecture of the IBM System/360. *IBM J. R&D*, 8(2):87, 1964.
- [8] AMI Semiconductor. Structured ASICs (accessed 28 September 07). http://www.amis.com/asics/structured_asics/index.html, 2007.
- [9] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller. Virtual simple architecture (VISA): exceeding the complexity limit in safe real-time systems. In *Proc. ISCA*, pages 350–361, New York, NY, USA, 2003. ACM Press.
- [10] A. Anantaraman, K. Seth, E. Rotenberg, and F. Mueller. Enforcing Safety of Real-Time Schedules on Contemporary Processors Using a Virtual Simple Architecture (VISA). In *Proc. RTSS*, pages 114–125, Washington, DC, USA, 2004. IEEE Computer Society.
- [11] Anonymous. Re: ASIC Economics (accessed 26 April 07). http://www.lightreading.com/boards/message.asp?msg_id=106551.
- [12] Apple Inc. Virtual PC for Mac OS X (accessed 26 April 07). <http://www.apple.com/macosex/applications/virtualpc/>.
- [13] ARC International. Home page (accessed 26 April 07). <http://www.arc.com/>.

-
- [14] ARM. ARMuLator. Application Note 32, ARM DAI 0032F, ARM Limited, 2003.
- [15] A. Arnaud and I. Puaut. Dynamic Instruction Cache Locking in Hard Real-Time Systems. In *Proc. RNTS*, Poitiers, France, May 2006.
- [16] R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding Worst-Case Instruction Cache Performance. In *Proc. RTSS*, pages 172–181, 1994.
- [17] P. J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.
- [18] ASIP Meister. Home page (accessed 26 April 07). <http://vlsilab.ics.es.osaka-u.ac.jp/dac2003/>.
- [19] N. Audsley and I. Bate. Synthesis of Legacy Real-Time Ada Software to FPGA. In *Proc. RTCSA*, pages 21–40, 2004.
- [20] N. Audsley, I. Bate, and M. Ward. Mapping Concurrent Real-Time Software to FPGA. In *Proceedings of the 3rd U.K. ACM SIGDA Workshop on Electronic Design Automation*, 2003.
- [21] N. Audsley, A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings. Fixed priority pre-emptive scheduling: an historical perspective. *Real-Time Syst.*, 8(2-3):173–198, 1995.
- [22] Avnet/Memec. Xilinx Spartan-3 Mini Module (accessed 23 January 08). http://www.em.avnet.com/ctf_shared/evk/df2df2usa/Xilinx_Spartan-3_Mini-Module-Product_Brief.pdf, 2006.
- [23] A. Balboni, W. Fornaciari, and D. Sciuto. Partitioning and Exploration Strategies in the TOSCA Co-Design Flow. In *Proc. CODES/CASHE*, page 62. IEEE Computer Society, 1996.
- [24] F. Barat, R. Lauwereins, and G. Deconinck. Reconfigurable instruction set processors from a hardware/software perspective. *IEEE Trans. Softw. Eng.*, 28(9):847–862, 2002.
- [25] M. Barr. Embedded Systems Glossary (accessed 28 September 07). <http://www.netrino.com/Publications/Glossary/>, 2003.
- [26] J. Barre, C. Landet, C. Rochange, and P. Sainrat. Modeling Instruction-Level Parallelism for WCET Evaluation. In *Proc. RTCSA*, pages 61–67, Washington, DC, USA, 2006. IEEE Computer Society.
- [27] J. Bartlett. The art of metaprogramming (accessed 26 April 07). <http://www-128.ibm.com/developerworks/linux/library/l-metaprogl.html>, 2005.
- [28] I. Bate, G. Bernat, and P. Puschner. Java virtual machine support for portable worst-case execution time analysis. In *ISORC*, Washington, USA, Jan 2002.
- [29] I. Bate and R. Reutemann. Efficient integration of bimodal branch prediction and pipeline analysis. In *Proc. RTCSA*, pages 39–44, Washington, DC, USA, 2005. IEEE Computer Society.
- [30] S. M. Bauer. Bell Labs microcode for the IBM 360/67. In *MICRO 8: Proceedings of the 8th annual workshop on Microprogramming*, pages 40–44, New York, NY, USA, 1975. ACM Press.

-
- [31] G. Bell. CDC 6600 registers (accessed 26 April 07). <http://research.microsoft.com/~gbell/craytalk/sld040.htm>.
- [32] F. Bellard. QEMU Open Source Processor Emulator (accessed 23 January 08). <http://fabrice.bellard.free.fr/qemu/>, 2007.
- [33] G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *25th IFAC Workshop on Real-Time Programming*, May 2000.
- [34] G. Bernat, A. Burns, and M. Newby. Probabilistic timing analysis: An approach using copulas. *J. Embedded Comput.*, 1(2):179–194, 2005.
- [35] G. Bernat, A. Colin, and S. M. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. In *Proc. RTSS*, pages 279–288, 2002.
- [36] V. Betz and J. Rose. VPR: A new packing, placement and routing tool for FPGA research. In *Proc. FPL*, pages 213–222. Springer-Verlag, 1997.
- [37] B. Blodget, S. McMillan, and P. Lysaght. A Lightweight Approach for Embedded Reconfiguration of FPGAs. In *Proc. DATE*, page 10399, Washington, DC, USA, 2003. IEEE Computer Society.
- [38] C. Bobda, M. Majer, A. Ahmadiania, T. Haller, A. Linarth, J. Teich, and J. van der Veen. The Erlangen Slot Machine: A Highly Flexible FPGA-Based Reconfigurable Platform. In *Proc. FCCM*, pages 319–320, Washington, DC, USA, 2005. IEEE Computer Society.
- [39] A. B. Bondi. Characteristics of scalability and their impact on performance. In *Proc. WOSP*, pages 195–203, New York, NY, USA, 2000. ACM Press.
- [40] M. Budiu and S. C. Goldstein. Fast compilation for pipelined reconfigurable fabrics. In *Proc. FPGA*, pages 195–205, New York, NY, USA, 1999. ACM Press.
- [41] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.
- [42] C. Burguiere and C. Rochange. A Contribution to Branch Prediction Modeling in WCET Analysis. In *Proc. DATE*, pages 612–617, Washington, DC, USA, 2005. IEEE Computer Society.
- [43] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, 2001.
- [44] J. Cantin and M. Hill. Cache performance for SPEC CPU2000 Benchmarks - Miss Ratio Tables (accessed 26 April 07). <http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data/>, 2003.
- [45] Celoxica Corporation. Handel-C: Software-Compiled System Design (accessed 26 April 07). <http://www.celoxica.com/>.
- [46] S. Chakraborty, S. Kunzli, and L. Thiele. Approximate schedulability analysis. In *Proc. RTSS*, page 159, Washington, DC, USA, 2002. IEEE Computer Society.

-
- [47] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. mei W. Hwu. IMPACT: an architectural framework for multiple-instruction-issue processors. In *Proc. ISCA*, pages 266–275, New York, NY, USA, 1991. ACM Press.
- [48] P. P. Chang, N. F. Warter, S. A. Mahlke, W. Y. Chen, and W. Hwu. Three architectural models for compiler-controlled speculative execution. *IEEE Trans. Comput.*, 44(4):481–494, 1995.
- [49] R. Chapman. *Static Timing Analysis and Program Proof*. PhD thesis, 1995.
- [50] R. Chapman, A. Burns, and A. Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Syst.*, 11(2):145–171, 1996.
- [51] A. E. Charlesworth. An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family. *IEEE Computer*, 14(9):18–27, 1981.
- [52] D. C. Chen and J. M. Rabaey. A Reconfigurable Multiprocessor IC for Rapid Prototyping of Algorithmic-Specific High-Speed DSP Data Paths. *IEEE Journal of Solid-State Circuits*, 27(12):1895–1904, 1992.
- [53] N. Cheung, J. Henkel, and S. Parameswaran. Rapid configuration and instruction selection for an ASIP: a case study. In *Proc. DATE*, page 10802, Washington, DC, USA, 2003. IEEE Computer Society.
- [54] J. M. Codina, J. Llosa, and A. Gonzalez. A comparative study of modulo scheduling techniques. In *Proc. ICS*, pages 97–106, New York, NY, USA, 2002. ACM Press.
- [55] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Syst.*, 18(2-3):249–274, 2000.
- [56] R. P. Colwell, R. P. Nix, J. J. O’Donnell, D. B. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proc. ASPLOS-II*, pages 180–192, 1987.
- [57] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.
- [58] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.
- [59] D. C. Cronquist, P. Franklin, C. Fisher, M. Figueroa, and C. Ebeling. Architecture design of reconfigurable pipelined datapaths. In *Proc. VLSI*, 1999.
- [60] J. C. Dehnert, P. Y.-T. Hsu, and J. P. Bratt. Overlapped loop support in the Cydra 5. In *Proc. ASPLOS*, pages 26–38, New York, NY, USA, 1989. ACM Press.
- [61] M. Delvai, W. Huber, P. Puschner, and A. Steininger. Processor Support for Temporal Predictability – The SPEAR Design Example. In *Proc. ECRTS*, Jul. 2003.
- [62] J. C. Demco and T. A. Marsland. An insight into PDP-11 emulation. In *Proc. MICRO*, pages 20–26, New York, NY, USA, 1976. ACM Press.
- [63] R. Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, August 2005.

-
- [64] J. Dongarra. An Overview of High Performance Computing (accessed 26 April 07). <http://www.netlib.org/utk/people/JackDongarra/SLIDES/dongarra-cetraro-2006.pdf>, 2004.
- [65] DOSBox Crew. DOSBox, x86 emulator with DOS (accessed 26 April 07). <http://dosbox.sf.net/>.
- [66] K. Ebcioglu and E. R. Altman. Daisy: dynamic compilation for 100% architectural compatibility. In *Proc. ISCA*, pages 26–37, 1997.
- [67] C. Ebeling. Compiling to Coarse-Grained Adaptable Architectures. Technical Report UW-CSE-02-06-01, University of Washington, 2002.
- [68] C. Ebeling. The General Rapid Architecture Description. Technical Report UW-CSE-02-06-02, University of Washington, 2002.
- [69] S. Edwards and E. A. Lee. The Case for the Precision Timed (PRET) Machine. Technical Report UCB/EECS-2006-149, EECS Department, University of California, Berkeley, Nov 2006.
- [70] EEMBC. The Embedded Microprocessor Benchmark Consortium (accessed 26 April 07). <http://www.eembc.org/>.
- [71] K. A. El-Ayat and J. A. Howard. Algorithms for a self-tuning microprogrammed computer. In *Proc. MICRO*, pages 85–91, Piscataway, NJ, USA, 1977. IEEE Press.
- [72] A. El-Haj-Mahmoud, A. S. AL-Zawawi, A. Anantaraman, and E. Rotenberg. Virtual multiprocessor: an analyzable, high-performance architecture for real-time computing. In *Proc. CASES*, pages 213–224, New York, NY, USA, 2005. ACM Press.
- [73] P. Eles, Z. Peng, K. Kuchcinski, and A. Daboli. System level hardware/software partitioning based on simulated annealing and tabu search. *Design Automation for Embedded Systems*, 2(1):5–32, 1997.
- [74] J. R. Ellis. *Bulldog: a compiler for vliw architectures (parallel computing, reduced-instruction-set, trace scheduling, scientific)*. PhD thesis, New Haven, CT, USA, 1985.
- [75] J. Engblom. Why specint95 should not be used to benchmark embedded systems tools. In *Proc. LCTES*, pages 96–103, 1999.
- [76] J. Engblom, P. Altenbernd, and A. Ermedahl. Facilitating worst-case execution time analysis for optimized code. In *ECRTS*, Berlin, Germany, June 1998.
- [77] Engineering Fundamentals. Least Square Method (accessed 18 October 07). http://www.efunda.com/math/least_squares/least_squares.cfm.
- [78] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. *LNCS*, 1300:1298–1307, 1997.
- [79] R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Des. Test*, 10(4):64–75, 1993.

-
- [80] H. Falk, S. Plazar, and H. Theiling. Compile-time decided instruction cache locking using worst-case execution paths. In *Proc. CODES+ISSS*, pages 143–148, New York, NY, USA, 2007. ACM Press.
- [81] P. Faraboschi, G. Desoli, and J. Fisher. Clustered Instruction-Level Parallel Processors. Technical Report HPL-98-204, HP Labs, 1998.
- [82] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proc. ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, pages 37–46, 1997.
- [83] G. Ferry. *A Computer Called LEO: Lyons Tea Shops and the World's First Office Computer*. HarperPerennial, 2004.
- [84] A. Filippov. Building an Ogg Theora camera using an FPGA and embedded Linux (accessed 26 April 07). <http://www.linuxdevices.com/articles/AT3888835064.html>, 2002.
- [85] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.*, 30(7):478–490, 1981.
- [86] J. Fisher. Very Long Instruction Word architectures and the ELI-512. In *Proc. Computer Architecture*, pages 140–150. IEEE Computer Society Press, 1983.
- [87] J. Fisher, P. Faraboschi, and C. Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2004.
- [88] C. W. Flink. EASY - an operating system for the QM-1. In *Proc. MICRO*, pages 7–14, Piscataway, NJ, USA, 1977. IEEE Press.
- [89] A. Fog. Branch prediction in the Pentium family (accessed 28 September 07). <http://www.x86.org/articles/branch/branchprediction.htm>.
- [90] W. Fornaciari and V. Piuri. Virtual FPGAs: Some steps behind the physical barriers. In *IPPS/SPDP Workshops*, pages 7–12, 1998.
- [91] N. Franklin. VirtexTools FPGA Programming and Debugging Tools Project (accessed 28 September 07). <http://neil.franklin.ch/Projects/VirtexTools/>, 2003.
- [92] Free On-Line Dictionary of Computing. Glue (accessed 18 October 07). <http://foldoc.org/?glue>.
- [93] Free Software Foundation. GCC Optimize Options (accessed 26 April 07). <http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>.
- [94] Free Software Foundation. GNU Compiler Collection (accessed 26 April 07). <http://gcc.gnu.org/>.
- [95] Free Software Foundation. Pragmas - Using the GNU Compiler Collection (GCC) (accessed 26 April 07). <http://gcc.gnu.org/onlinedocs/gcc/Pragmas.html>.
- [96] Free Software Foundation. i386.md: GCC 3.4.3 machine description for IA-32 and x86-64 (accessed 26 April 07). <http://gcc.gnu.org/gcc-3.4/>, 2004.

-
- [97] Gaisler Research. LEON-2 and LEON-3 Processors (accessed 23 January 08). <http://www.gaisler.com/leonmain.html>, 2007.
- [98] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. SpecsSyn: an environment supporting the specify-explore-refine paradigm for hardware/software system design. *Readings in hardware/software co-design*, pages 108–124, 2002.
- [99] P. Genua. A Cache Primer (accessed 26 April 07). http://www.freescale.com/files/32bit/doc/app_note/AN2663.pdf, 2004.
- [100] GNU Project. GNU Linear Programming Kit (accessed 23 January 08). <http://www.gnu.org/software/glpk/>, 2007.
- [101] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor. PipeRench: A reconfigurable architecture and compiler. *Computer*, 33(4):70–77, 2000.
- [102] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. Piperench: A coprocessor for streaming multimedia acceleration. In *Proc. ISCA*, pages 28–39, 1999.
- [103] R. E. Gonzalez. Xtensa — A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, 2000.
- [104] R. Govindarajan, E. R. Altman, and G. R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proc. MICRO*, pages 85–94, New York, NY, USA, 1994. ACM Press.
- [105] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. The benefits and costs of DyC’s run-time optimizations. *ACM Trans. Program. Lang. Syst.*, 22(5):932–972, 2000.
- [106] B. Grattan, G. Stitt, and F. Vahid. Codesign-extended applications. In *Proc. 10th Int. Symp. Hardware/Software Codesign*, pages 1–6, 2002.
- [107] R. K. Gupta and G. D. Micheli. Hardware-software cosynthesis for digital systems. *IEEE Des. Test*, 10(3):29–41, 1993.
- [108] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proc. IISWC*, 2001.
- [109] N. A. Harman. High Performance Microprocessors, 3rd Edition: Example: Intel IA64 and Itanium (accessed 23 January 08). <http://cs.swan.ac.uk/~csneal/HPM/merced.html>, 2005.
- [110] R. Hartenstein. Coarse grain reconfigurable architectures. In *Proc. ASP-DAC*, pages 564–570, 2001.
- [111] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. Using the KressArray for Reconfigurable Computing. In *Proc. SPIE*, pages 150–161, Bellingham, WA, November 1998.
- [112] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. KressArray Xplorer: a new CAD environment to optimize reconfigurable datapath array. In *Proc. ASP-DAC*, pages 163–168, New York, NY, USA, 2000. ACM Press.

-
- [113] R. W. Hartenstein and R. Kress. A datapath synthesis system for the reconfigurable datapath architecture. In *Proc. ASP-DAC*, page 77, New York, NY, USA, 1995. ACM Press.
- [114] Haskell.org. Introduction (accessed 23 January 08). <http://www.haskell.org/haskellwiki/Introduction>, 2007.
- [115] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera reconfigurable functional unit. In *Proc. FCCM*, page 87, 1997.
- [116] J. R. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. In *Proc. FCCM*, page 12. IEEE Computer Society, 1997.
- [117] W. Havanki, S. Banerjia, and T. Conte. Treeregion scheduling for wide issue processors. In *Proc. HPCA*, page 266, Washington, DC, USA, 1998. IEEE Computer Society.
- [118] B. Hayes. Differences in Optimizing for the Pentium 4 Processor vs. the Pentium III Processor (accessed 26 April 07). <http://www.intel.com/cd/ids/developer/asm-na/eng/44010.htm?page=4>.
- [119] C. A. Healy, R. D. Arnold, F. Mueller, M. G. Harmon, and D. B. Whalley. Bounding pipeline and instruction cache performance. *IEEE Trans. Comput.*, 48(1):53–70, 1999.
- [120] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Proc. RTSS*, page 288, Washington, DC, USA, 1995. IEEE Computer Society.
- [121] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proc. IEEE*, 91(7):1038–1054, 2003.
- [122] J. Henkel and R. Ernst. A hardware/software partitioner using a dynamically determined granularity. In *Proc. DAC*, pages 691–696, 1997.
- [123] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.
- [124] R. B. Hilgendorf, G. J. Heim, and W. Rosenstiel. Evaluation of branch-prediction methods on traces from commercial applications. *IBM J. R&D*, 43(4):579–593, 1999.
- [125] M. D. Hill. A case for direct-mapped caches. *Computer*, 21(12):25–40, 1988.
- [126] M. Hipp and T. Orgis. mpg123 - Fast console MPEG Audio Player (accessed 26 April 07). <http://www.mpg123.de/>.
- [127] P. Horowitz and W. Hill. *The Art of Electronics*. Cambridge University Press, New York, NY, USA, 1989.
- [128] E. L. Horta, J. W. Lockwood, D. E. Taylor, and D. Parlour. Dynamic hardware plugins in an FPGA with partial run-time reconfiguration. In *Proc. DAC*, pages 343–348. ACM Press, 2002.

-
- [129] M. Hubner and J. Becker. Exploiting dynamic and partial reconfiguration for FPGAs: toolflow, architecture and system integration. In *Proc. SBCCI*, pages 1–4, New York, NY, USA, 2006. ACM Press.
- [130] S. Huss-Lederman, E. M. Jacobson, A. Tsao, T. Turnbull, and J. R. Johnson. Implementation of strassen’s algorithm for matrix multiplication. In *Supercomputing*, page 32, Washington, DC, USA, 1996. IEEE Computer Society.
- [131] IBM Archives. Automatic Sequence Controlled Calculator - Feeds, speeds and specifications (accessed 26 April 07). http://www-03.ibm.com/ibm/history/exhibits/markI/markI_feeds.html.
- [132] ILOG. CPLEX: High-performance software for mathematical programming (accessed 23 January 08). <http://www.ilog.com/products/cplex/>, 2007.
- [133] Intel. Intel Architecture Optimization Manual (accessed 26 April 07). <ftp://download.intel.com/design/PentiumII/manuals/24281603.PDF>, 1997.
- [134] Intel. Optimizing Applications with the Intel C++ and Fortran Compilers (accessed 26 April 07). ftp://download.intel.com/software/products/compilers/techttopics/Compiler_Optimization_7_02.pdf, 2004.
- [135] Intel. Intel Pentium 4 Processors For Embedded Computing (accessed 23 January 08). <http://www.intel.com/design/intarch/pentium4/pentium4.htm>, 2007.
- [136] Intel. Intel Pentium Processors with MMX Technology for Embedded Computing (accessed 23 January 08). <http://www.intel.com/design/intarch/mmx/mmx.htm>, 2007.
- [137] Intel. Intel’s First Microprocessor – the Intel 4004 (accessed 23 January 08). <http://www.intel.com/museum/archives/4004.htm>, 2007.
- [138] Intel Technology Journal. Hyper Threading Technology (accessed 26 April 07). ftp://download.intel.com/technology/itj/2002/volume06issue01/vol6iss1_hyper_threading_technology.pdf, 2002.
- [139] A. Janapsatya, A. Ignjatovic, and S. Parameswaran. Exploiting statistical information for implementation of instruction scratchpad memory in embedded system. *IEEE Trans. VLSI*, 14(8):816–829, 2006.
- [140] M. T. Jones. Optimization in GCC (accessed 26 April 07). <http://www.linuxjournal.com/article/7269>, 2005.
- [141] M. Joseph and P. K. Pandya. Finding response times in a real-time system. *Comput. J.*, 29(5):390–395, 1986.
- [142] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. R&D*, 49(4/5):589–604, 2005.

-
- [143] A. Kalavade and E. A. Lee. A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem. In *Proc. CODES*, pages 42–48. IEEE Computer Society Press, 1994.
- [144] M. G. H. Katevenis. *Reduced instruction set computer architectures for VLSI*. Massachusetts Institute of Technology, Cambridge, MA, USA, 1985.
- [145] D. B. Kirk. Smart (strategic memory allocation for real-time) cache design. In *Proc. RTSS*, pages 229–237, 1989.
- [146] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220, 4598:671–680, 1983.
- [147] C. P. Kruskal and E. Weixelbaum. A note on the worst case of heapsort. In *SIGACT News*, volume 13, pages 35–38, New York, NY, USA, 1981. ACM Press.
- [148] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis. Loading $\rho\mu$ -code: Design considerations. In *Proc. SAMOS*, pages 11–19, July 2003.
- [149] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis. The Molen Media Processor: Design and Evaluation. In *Proc. WASP*, pages 26–33, September 2005.
- [150] D. Lampret. OpenRISC 1200 (accessed 26 April 07). <http://www.opencores.org/>.
- [151] D. Lampret. OpenRISC 1000 Architecture Manual (accessed 26 April 07). http://www.opencores.org/cvsget.cgi/or1k/docs/openrisc_arch.pdf, 2006.
- [152] D. Landskov, S. Davidson, B. Shriver, and P. W. Mallett. Local microcode compaction techniques. *ACM Comput. Surv.*, 12(3):261–294, 1980.
- [153] K. P. Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux Journal*, 1996(29es):7, 1996.
- [154] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Int. Symp. Microarchitecture*, pages 330–335, 1997.
- [155] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.*, 47(6):700–713, 1998.
- [156] M. Leone and R. K. Dybvig. Dynamo: A Staged Compiler Architecture for Dynamic Program Optimisation. Technical Report 490, Dept. of Computer Science, Indiana University, 1997.
- [157] T. G. Lewis and B. D. Shriver. Introduction. *IEEE Trans. Comput.*, 30(7):457–459, 1981.
- [158] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Syst.*, 34(3):195–227, 2006.
- [159] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proc. DAC*, pages 456–461, New York, NY, USA, 1995. ACM Press.

-
- [160] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *Proc. RTSS*, page 254, Washington, DC, USA, 1996. IEEE Computer Society.
- [161] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Code optimization techniques for embedded DSP microprocessors. In *Proc. DAC*, pages 599–604, New York, NY, USA, 1995. ACM Press.
- [162] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C. S. Kim. An Accurate Worst Case Timing Analysis for RISC Processors. *IEEE Trans. Softw. Eng.*, 21(7):593–604, 1995.
- [163] S.-S. Lim, J. H. Han, J. Kim, and S. L. Min. A worst case timing analysis technique for multiple-issue machines. In *Proc. RTSS*, pages 334–345, 1998.
- [164] Linux MIPS. AR7 (accessed 18 October 07). <http://www.linux-mips.org/wiki/AR7>.
- [165] J. Llosa, A. Gonzalez, E. Ayguade, and M. Valero. Swing Modulo Scheduling: A Lifetime-Sensitive Approach. In *Proc. PACT*, page 80, Washington, DC, USA, 1996. IEEE Computer Society.
- [166] Lpsolve Team. SourceForge.net: lpsolve (accessed 23 January 08). <http://sourceforge.net/projects/lpsolve>, 2007.
- [167] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Proc. RTSS*, page 12. IEEE Computer Society, 1999.
- [168] R. Lysecky, G. Stitt, and F. Vahid. Warp processors. *ACM TODAES*, 11(3):659–681, 2006.
- [169] D. Macos and F. Mueller. Integrating Gnat/Gcc into a Timing Analysis Environment (WIP). In *Proc. EUROMICRO*, 1998.
- [170] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proc. MICRO*, pages 45–54, 1992.
- [171] Malardalen WCET research group. WCET Benchmarks (accessed 18 October 07). <http://www.mrtc.mdh.se/projects/wcet/benchmarks.html>, 2007.
- [172] R. Marlet, C. Consel, and P. Boinot. Efficient incremental run-time specialization for free. In *Proc. PLDI*, pages 281–292, New York, NY, USA, 1999. ACM Press.
- [173] S. McMillan and C. Patterson. JBits(TM) Implementations of the Advanced Encryption Standard (Rijndael). In *Proc. FPL*, pages 162–171, London, UK, 2001. Springer-Verlag.
- [174] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins. Design Methodology for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture: A Case Study. In *Proc. DATE*, page 21224. IEEE Computer Society, 2004.
- [175] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. Exploiting Loop-Level Parallelism on Coarse-Grained Reconfigurable Architectures Using Modulo Scheduling. In *Proc. DATE*, page 10296. IEEE Computer Society, 2003.

-
- [176] M. C. Merten, A. R. Trick, and R. D. Barnes. An architectural framework for runtime optimization. *IEEE Trans. Comput.*, 50(6):567–589, 2001.
- [177] Microsoft Support. A microcode reliability update is available that improves the reliability of systems that use Intel processors (accessed 23 January 08). <http://support.microsoft.com/kb/936357>, 2007.
- [178] E. Mirsky and A. DeHon. MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. In K. L. Pocek and J. Arnold, editors, *Proc. FCCM*, pages 157–166, Los Alamitos, CA, 1996. IEEE Computer Society Press.
- [179] T. Miyamori and K. Olukotun. REMARC: Reconfigurable multimedia array coprocessor. In *FPGA*, page 261, 1998.
- [180] R. Mosher. FPGA Prototyping to Structured ASIC Production to Reduce Cost, Risk and TTM (accessed 28 September 07). <http://www.us.design-reuse.com/articles/article13550.html>.
- [181] Motorola Inc. *MC68000 16-bit Microprocessor User's Manual 3rd Edition*. Prentice Hall, 1982.
- [182] F. Mueller. Compiler support for software-based cache partitioning. In *Proc. LCTES*, pages 125–133, New York, NY, USA, 1995. ACM Press.
- [183] F. Mueller. Timing analysis for instruction caches. *Real-Time Syst.*, 18(2-3):217–247, 2000.
- [184] D. R. Musser. Introsort (accessed 26 April 07). <http://www.cs.rpi.edu/~musser/gp/introsort.ps>.
- [185] J. Nakajima. Re: [PATCH] Fixes for building kernel using Intel compiler (accessed 26 April 07). <http://www.uwsg.iu.edu/hypermail/linux/kernel/0210.3/1025.html>, 2002.
- [186] A. Nicolau and J. A. Fisher. Using an oracle to measure potential parallelism in single instruction stream programs. In *Proc. MICRO*, pages 171–182, Piscataway, NJ, USA, 1981. IEEE Press.
- [187] NIST. Advanced Encryption Standard - Questions and Answers (accessed 26 April 07). <http://csrc.nist.gov/encryption/aes/aesfact.html>, 2002.
- [188] P. Norton. *The Peter Norton programmer's guide to the IBM PC*. Microsoft Press, Redmond, WA, USA, 1985.
- [189] Opencores.org. Opencores.org home page (accessed 26 April 07). <http://www.opencores.org/>.
- [190] Opencores.org. Wishbone Bus Specification (accessed 23 January 08). http://www.opencores.org/projects.cgi/web/wishbone/wbspec_b3.pdf, 2007.
- [191] R. J. Pankhurst. Operating systems: Program overlay techniques. *Commun. ACM*, 11(2):119–125, 1968.

-
- [192] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Syst.*, 5(1):31–62, 1993.
- [193] N. Patavalis. A Brief Introduction to the JTAG Boundary Scan Interface (accessed 23 January 08). <http://www.inaccessnetworks.com/projects/ianjtag/jtag-intro/jtag-intro.html>, 2001.
- [194] H. Patil and J. Emer. Combining static and dynamic branch prediction to reduce destructive aliasing. In *Proc. HPCA*, pages 251–252, 2000.
- [195] D. A. Patterson and J. L. Hennessy. *Computer organization & design: the hardware/software interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.
- [196] B. Patwardhan. Introduction to the Streaming SIMD Extensions in the Pentium III (accessed 23 January 08). http://www.x86.org/articles/sse_pt1/simd1.htm.
- [197] PetaLogix. Linux Solutions (accessed 23 January 08). <http://www.petalogix.com/>, 2007.
- [198] B. Pfaffenberger. Why is Microsoft Attacking the GPL? - IBM vs. Amdahl (accessed 23 January 08). <http://gnu.org.in/pipermail/fsf-india/2001-August/001697.html>, 2001.
- [199] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. ‘C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, March 1999.
- [200] I. Puaut. Cache analysis vs static cache locking for schedulability analysis in multitasking real-time systems. In *Proc. WCET*, Vienna, Austria, June 2002.
- [201] I. Puaut and D. Hardy. Predictable paging in real-time systems: A compiler approach. In *Proc. ECRTS*, pages 169–178, Washington, DC, USA, 2007. IEEE Computer Society.
- [202] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proc. DATE*, pages 1484–1489, San Jose, CA, USA, 2007. EDA Consortium.
- [203] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. Technical Report PI 1818, IRISA, 2007.
- [204] P. Puschner. Is worst-case execution-time analysis a non-problem? – towards new software and hardware architectures. In *Proc. ECRTS*, Technical Report, York YO10 5DD, United Kingdom, Jun. 2002. Department of Computer Science, University of York.
- [205] P. Puschner and A. Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Syst.*, 18(2-3):115–128, 2000.
- [206] P. Puschner and C. Koza. Calculating the maximum, execution time of real-time programs. *Real-Time Syst.*, 1(2):159–176, 1989.
- [207] P. Puschner and A. Schedl. Computing maximum task execution times - a graph-based approach. *Real-Time Syst.*, 13(1):67–91, 1997.

-
- [208] Python Software Foundation. Python Official Website (accessed 26 April 07). <http://python.org/>.
- [209] R. Niemann and P. Marwedel. Hardware/software partitioning using integer programming. In *Proceedings of the European Design and Test Conference (ED & TC)*, pages 473–480, Paris, France, 1996. IEEE Computer Society Press (Los Alamitos, California).
- [210] F. Ramirez. *Automated Conversion from LUT-based FPGAs to LUT-based MPGAs*. PhD thesis, Ulm, Germany, 2007.
- [211] B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proc. MICRO*, pages 63–74, New York, NY, USA, 1994. ACM Press.
- [212] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proc. MICRO*, pages 183–198, Piscataway, NJ, USA, 1981. IEEE Press.
- [213] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proc. MICRO*, pages 172–180, New York, NY, USA, 1994. ACM Press.
- [214] J. Regehr. *Using Hierarchical Scheduling to Support Soft Real-Time Applications on General-Purpose Operating Systems*. PhD thesis, University of Virginia, 2001.
- [215] M. Reshadi and D. Gajski. A cycle-accurate compilation algorithm for custom pipelined datapaths. In *Proc. CODES/ISSS*, pages 21–26, 2005.
- [216] A. Rigo. Representation-based just-in-time specialization and the Psycho prototype for Python. In *Proc. PEPM*, pages 15–26. ACM Press, 2004.
- [217] C. Rochange and P. Sainrat. A time-predictable execution mode for superscalar pipelines with instruction prescheduling. In *Proc. CF*, pages 307–314, New York, NY, USA, 2005. ACM Press.
- [218] R. F. Rosin, G. Frieder, and J. Richard H. Eckhouse. An environment for research in micro-programming and emulation. *Commun. ACM*, 15(8):748–760, 1972.
- [219] J. E. Roskos and R. I. Winner. Toward user sharing of the microprogramming level under UNIX on the Perkin-Elmer 3220. In *Proc. MICRO*, pages 67–73, Piscataway, NJ, USA, 1981. IEEE Press.
- [220] E. Rotenberg. Trace caches. In *Speculative Execution in High Performance Computer Architectures*, pages 87–108. CRC Press, 2005.
- [221] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proc. MICRO*, pages 24–35, Washington, DC, USA, 1996. IEEE Computer Society.
- [222] B. Sanderson. RAM, Virtual Memory, Pagefile and all that stuff (accessed 26 April 07). <http://support.microsoft.com/kb/555223>.

-
- [223] M. S. Schlansker and B. R. Rau. EPIC: An Architecture for Instruction-Level Parallel Processors. Technical Report HPL-1999-111, HP Labs, 1999.
- [224] H. Schmit, D. Whelihan, M. Moe, B. Levine, and R. Taylor. PipeRench: A virtualized programmable datapath. In *Proc. CICC*, pages 63–66, 2002.
- [225] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1993.
- [226] D. Sciuto, S. Antoniazzi, A. Balboni, and W. Fornaciari. The role of VHDL within the TOSCA hardware/software codesign framework. In *Proc. European Design Automation*, pages 612–617. IEEE Computer Society Press, 1994.
- [227] L. Semiconductor. 3.3v universal pld. Data sheet GAL22LV10, 2005.
- [228] C. H. Sequin and D. A. Patterson. Design and Implementation of Risc-1. Technical Report CSD-82-106, Computer Science Division, UC Berkeley, 1982.
- [229] SGI. Standard Template Library Programmer’s Guide (accessed 26 April 07). <http://www.sgi.com/tech/stl/>.
- [230] G. Shvets. Intel 8086 microprocessor family (accessed 23 January 08). <http://www.cpu-world.com/CPUs/8086/>, 2003.
- [231] SimpleScalar LLC. SimpleScalar Hacker’s guide (accessed 26 April 07). http://www.simplescalar.com/docs/hack_guide_v2.pdf.
- [232] SimpleScalar LLC. SimpleScalar Tutorial (accessed 26 April 07). http://www.simplescalar.com/docs/simple_tutorial_v4.pdf.
- [233] J. E. Smith. A study of branch prediction strategies. In *Proc. ISCA*, pages 135–148, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.
- [234] J. E. Smith and A. R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proc. ISCA*, pages 36–44, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.
- [235] M. D. Smith, M. S. Lam, and M. A. Horowitz. Boosting beyond static scheduling in a superscalar processor. *SIGARCH Comput. Archit. News*, 18(3a):344–354, 1990.
- [236] G. S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Trans. on Computers*, 39(3):349–359.
- [237] SPARC International. The SPARC Architecture Manual: Version 8 (accessed 18 October 07). <http://gaisler.com/doc/sparcv8.pdf>, 1992.
- [238] J. A. Stankovic. The types and interactions of vertical migration of functions in a multi-level interpretive system. *IEEE Trans. Computers*, 30(7), 1981.
- [239] J. A. Stankovic and K. Ramamritham. The Spring kernel: a new paradigm for real-time operating systems. *SIGOPS Oper. Syst. Rev.*, 23(3):54–71, 1989.

-
- [240] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *Proc. ISSS*, pages 213–218, New York, NY, USA, 2002. ACM Press.
- [241] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proc. DATE*, page 409, Washington, DC, USA, 2002. IEEE Computer Society.
- [242] J. Stockenberg and A. van Dam. Vertical migration for performance enhancement in layered hardware/firmware/software systems. *Computer*, 11(5), 1981.
- [243] M. Stodte. DAISY: an open-source JIT compiler for large machines (accessed 26 April 07). <http://www-128.ibm.com/developerworks/library/os-daisy.html>, 2000.
- [244] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET Centric Data Allocation to Scratchpad Memory. In *Proc. RTSS*, pages 223–232, Washington, DC, USA, 2005. IEEE Computer Society.
- [245] Sun Microsystems. Java Hotspot Server VM: Dynamic Compilation (accessed 26 April 07). <http://java.sun.com/products/hotspot/docs/general/hs2.html>.
- [246] R. A. Sutton, V. P. Srini, and J. M. Rabaey. A multiprocessor DSP system using PADDI-2. In *Proc. DAC*, pages 62–65, New York, NY, USA, 1998. ACM Press.
- [247] Tensilica. XPRES Triple Threat Solution (accessed 26 April 07). http://www.tensilica.com/pdf/XPRES-Triple-Threat_Solution.pdf.
- [248] Tensilica. Xtensa Wins EEMBC Benchmarks (accessed 26 April 07). http://www1.tensilica.com/news_events/pr_2002_08_26a.htm.
- [249] R. G. Tessier. *Fast Place and Route Approaches for FPGAs*. PhD thesis, 1999.
- [250] Texas Instruments. AR7 DSL Set-Top Box Development Board Solution (accessed 18 October 07). <http://focus.ti.com/lit/ml/spat150a/spat150a.pdf>.
- [251] H. Theiling and C. Ferdinand. Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis. In *Proc. RTSS*, page 144, Washington, DC, USA, 1998. IEEE Computer Society.
- [252] R. T. Thomas. The development of user microprogramming: A survey and status report. In *Proc. MICRO*, pages 212–216, New York, NY, USA, 1974. ACM Press.
- [253] M. Tokoro, E. Tamura, and T. Takizuka. Optimization of microprograms. *IEEE Trans. Comput.*, 30(7):491–504, 1981.
- [254] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. R&D*, 11(1):25–33, 1967.
- [255] Transmeta Corporation. Home page (accessed 26 April 07). <http://www.transmeta.com/>.

-
- [256] Transmeta Corporation. Efficeon Processor (accessed 26 April 07). <http://www.transmeta.com/efficeon/index.html>, 2007.
- [257] N. Tredennick. Experiences in commercial VLSI microprocessor design. *Microproc. Microsyst.*, 12(8):419–432, 1988.
- [258] S. G. Tucker. Microprogram control for System/360. *IBM Systems Journal*, 6(4):222–241, 1967.
- [259] M. Ullmann, M. Hubner, B. Grimm, and J. Becker. An FPGA Run-Time System for Dynamical On-Demand Reconfiguration. *IPDPS*, 04:135a, 2004.
- [260] M. Ullmann, M. Hbner, B. Grimm, and J. Becker. On-Demand FPGA Run-Time System for Dynamical Reconfiguration with Adaptive Priorities. In *Proc. FPL*, 2004.
- [261] USPTO. US Patent 3400379 (accessed 28 September 07). <http://www.freepatentsonline.com/3400379.html>.
- [262] F. Vahid. Modifying Min-Cut for Hardware and Software Functional Partitioning. In *Proc. CODES*, page 43. IEEE Computer Society, 1997.
- [263] G. Vanmeerbeeck, P. Schaumont, S. Vernalde, M. Engels, and I. Bolsens. Hardware/software partitioning of embedded system in OCAPI-xl. In *Proc. CODES*, pages 30–35. ACM Press, 2001.
- [264] N. Vassiliadis, N. Kavvadias, G. Theoridis, and S. Nikolaidis. A RISC architecture extended by an efficient tightly coupled reconfigurable unit. *International Journal of Electronics*, 93:421–438(18), June 2006.
- [265] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte. The Molen Polymorphic Processor. *IEEE Trans. Comput.*, pages 1363–1375, November 2004.
- [266] Verilog.com. Verilog Resources (accessed 26 April 07). <http://www.verilog.com/>.
- [267] H. Walder and M. Platzner. A runtime environment for reconfigurable hardware operating systems. In *Proc. FPL*, pages 831–835, 2004.
- [268] D. W. Wall. Limits of Instruction-Level Parallelism. Technical Report WRL-93-6, DEC Western Research Laboratory, 1995.
- [269] D. Wallner. T80 CPU Core (accessed 23 January 08). <http://www.opencores.org/projects/t80/>, 2007.
- [270] A. Wang, E. Killian, D. Maydan, and C. Rowen. Hardware/software instruction set configurability for system-on-chip processors. In *Proc. DAC*, pages 184–188, 2001.
- [271] M. Ward. *Improving the Timing Analysis of Ravenscar/SPARK Ada by Direct Compilation to Hardware*. PhD thesis, 2005.
- [272] M. Ward and N. Audsley. Language issues of compiling Ada to hardware. *Ada Lett.*, XXII(4):85–94, 2002.

-
- [273] L. Wehmeyer and P. Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *Proc. DATE*, pages 600–605, Washington, DC, USA, 2005. IEEE Computer Society.
- [274] E. Weisstein. Set Partition, From MathWorld—A Wolfram Web Resource. <http://mathworld.wolfram.com/SetPartition.html>.
- [275] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Proc. Int. Conf. Quality Software*, Sep. 2005.
- [276] J. Whitham and N. Audsley. MCGREP - A Predictable Architecture for Embedded Real-time Systems. In *Proc. RTSS*, pages 13–24, 2006.
- [277] J. Whitham and N. Audsley. A self-optimising simulator for a coarse-grained reconfigurable array. In *Proc. UK Embedded Forum*, pages 99–109. University of Newcastle, April 2007.
- [278] J. Whitham and N. Audsley. Using trace scratchpads to reduce execution times in predictable real-time architectures. In *Proc. RTAS*, 2008.
- [279] G. Wigley, D. Kearney, and D. Warren. Introducing ReConfigME: An Operating System for Reconfigurable Computing. In *Proc. FPL*, pages 687–697, London, UK, 2002. Springer-Verlag.
- [280] R. Wilhelm. Why AI + ILP Is Good for WCET, but MC Is Not, Nor ILP Alone. *LNCS*, 2937:309–322, 2004.
- [281] Xilinx. Partial Reconfigurability FAQ (accessed 26 April 07). http://www.xilinx.com/products/design_tools/logic_design/advanced/partial_reconf_faq.htm.
- [282] Xilinx. Embedded processing and control solutions for Spartan-3 FPGAs. Application Note XAPP477, Xilinx Corporation, 2003.
- [283] Xilinx. Spartan-IIIE 1.8V FPGA Family. Datasheet DS077, Xilinx Corporation, 2004.
- [284] Xilinx. Two Flows for Partial Reconfiguration: Module Based or Difference Based. Application Note XAPP290, Xilinx Corporation, 2004.
- [285] Xilinx. Virtex-II platform FPGAs. Datasheet DS031, Xilinx Corporation, 2004.
- [286] Xilinx. Microblaze processor reference guide. Manual UG081, Xilinx Corporation, 2005.
- [287] Xilinx. Configuration Solutions for Spartan-3A (accessed 26 April 07). http://www.xilinx.com/products/design_resources/config_sol/s3/config_s3e.htm, 2007.
- [288] Xilinx. FPGA vs. ASIC (accessed 23 January 08). <http://www.xilinx.com/company/gettingstarted/fpgavsasic.htm>, 2007.
- [289] Xilinx. ML310 User Guide: Virtex-II Embedded Development Platform. Manual UG068, Xilinx Corporation, 2007.
- [290] Xilinx. Virtex-4 Family Overview. Datasheet DS112, Xilinx Corporation, 2007.

-
- [291] Xilinx. XC2C512 CoolRunner-II CPLD. Data Sheet DS096, Xilinx Corporation, 2007.
- [292] X. H. Xu, C. T. Clarke, and J. T. Taylor. Dual-Huffman Based Code Compression For Embedded ARM/Thumb Processors. In *Proc. UK Embedded Forum*, pages 59–69. University of Newcastle, April 2007.
- [293] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proc. 27th Int. Symp. Computer Architecture*, pages 225–235, 2000.
- [294] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *Real-Time Syst.*, 5(4):319–343, 1993.
- [295] W. Zhao, W. Krehling, D. Whalley, C. Healy, and F. Mueller. Improving WCET by applying worst-case path optimizations. *Real-Time Syst.*, 34(2):129–152, 2006.
- [296] H. Zhou, M. Jennings, and T. Conte. Tree Traversal Scheduling: A Global Scheduling Technique for VLIW/EPIC Processors. In *Proc. LCPC*. Springer Verlag, 2001.

Appendix A

Digital Appendix Documentation

This appendix refers to the software in the companion data archive bundled with this thesis. If this copy of the thesis does not include this archive, it can be downloaded from the following location:

<http://www.jwhitham.org.uk/thesis/>

The layout of this appendix is as follows:

- Section A.1 is an overview of the programs in the archive.
- Additional software required (or recommended) for use with the archive is listed in section A.2.
- General installation advice is given in section A.3.
- Sections A.4 to A.12 describe the programs in the archive: see section A.1 for an overview.
- The third party software included in the distribution is discussed in section A.13.

A.1 Overview

The software archive is a collection of research programs implementing the experiments in chapters 5 to 8. There is no integrated development environment (IDE). Instead, the programs of the Appendix are organised into separate projects, as illustrated in Figure A.1. The projects are:

- `/mcgrep1`: experiments from chapter 5 including the MCGREP-1 CPU generator. These programs are described in sections A.4 to A.6.
- `/testcases`: source code for the MCGREP-2 experiments in section 6.4. The programs are described in section A.7. Benchmark code is written in C, with a test infrastructure written in Python.
- `/mcgrep2-src`: the MCGREP-2 CPU generator, simulator, and tools for compiling and debugging programs, as described in chapter 6. These programs are described in sections A.8 to A.10, and are written in Python. C and VHDL templates are used for code generation.
- `/tracegen`: the trace generator, which is written in C. The trace generator reads trace information from machine code and generates a suitable microprogram. It is specialised at compile time for one MCGREP-2 CPU via the microprogramming API (section 6.2.4). The

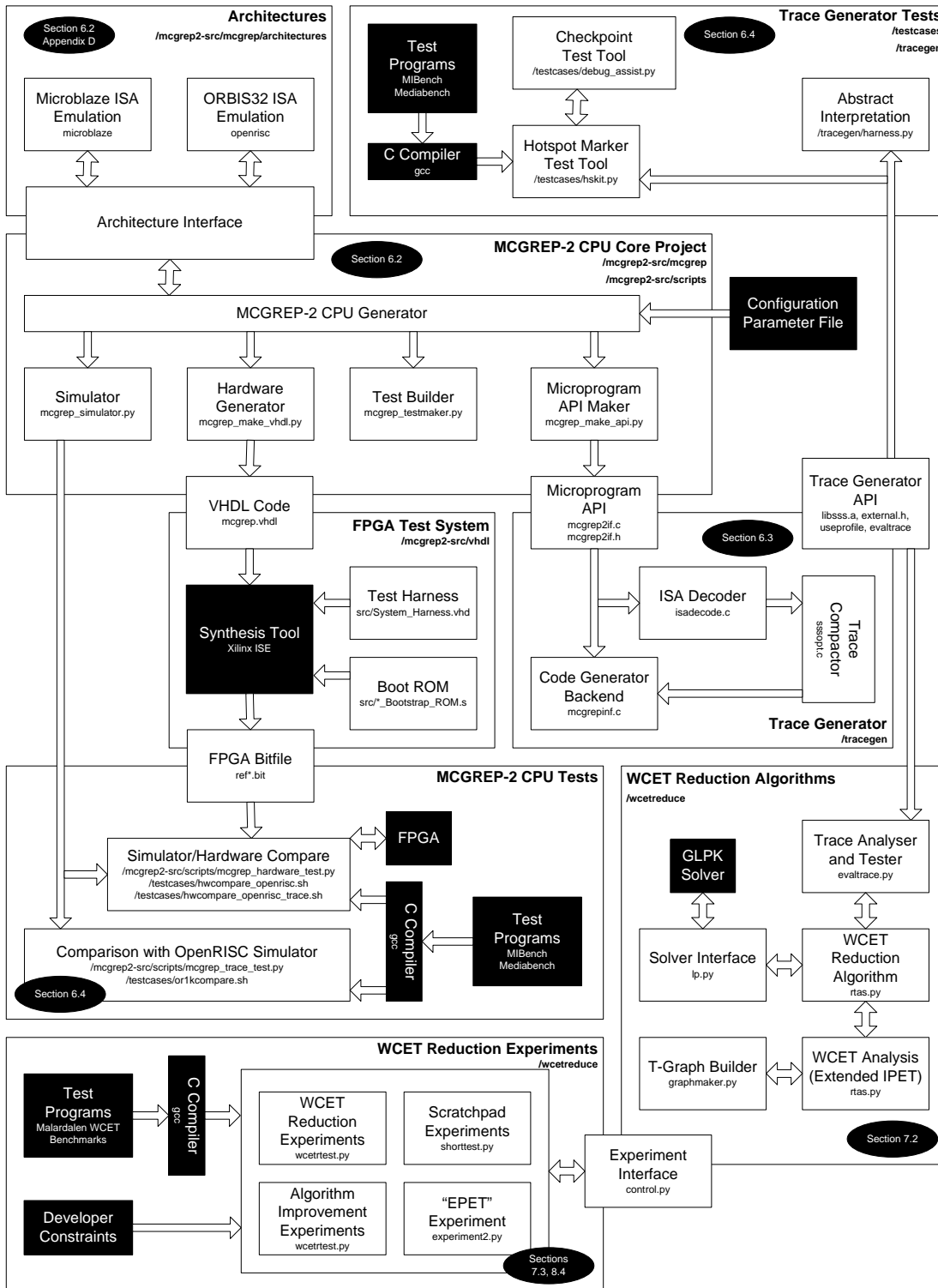


Figure A.1: A map of the software described by chapters 6 through 8, showing file and path names within the archive.

`useprofile` and `evaltrace` tools are used by the experiments described in section 6.3.1 and 7.2.7 respectively.

- `/wcetreduce`: experiment source code for chapters 7 and 8.

Your programs can use MCGREP-2's microprogramming features by linking against the `tracegen` library `libsss.a` built for the MCGREP-2 architecture configuration you wish to use. Sample usages of the API for direct microprogramming from C can be found in the `/tracegen-/mcgrepinf.c` file. You can also rerun the tests used to check MCGREP-2 as described in section A.12. Some of these tests require an FPGA: information about the required arrangement can be found in section A.11. Please note that the tests are not all "task safe": some tests may fail if run in parallel with each other on the same filesystem, and FPGA access is assumed to be exclusive.

A.2 Required/recommended Software Environment

The archive was prepared on an x86-compatible computer running the Debian Etch Linux distribution. It is also known to be compatible with x86 computers running Slackware Linux version 12.0.0.

In principle, the archive software should be usable on any computer but in practice, some components need a Linux-like environment. Support for x86 Linux binaries is required to support the `gcc` cross compiler for ORBIS32, although this software can be rebuilt using the bundled source code (section A.13). A number of additional programs are required for most operations. These should be provided by the operating system:

- A working `gcc` installation for the host computer (i.e. a version of `gcc` that generates native binaries for the host).

Recommended versions: `gcc` versions 3.4.6, 4.1.2 and 4.1.3 have been tested.

- A working environment for building C programs (i.e. all header files installed).

Recommended versions: `libc6` versions 2.3.6 and 2.6.1 have been tested.

- Python, with support for building C extensions.

Recommended versions: Python 2.4.4 or 2.5.1.

This command could be used on Debian Linux to install the required components:

```
apt-get install python bzip2 python-dev libc6-dev gcc
```

The following additional programs are recommended, because they are required by some operations:

- Xilinx Integrated Software Environment (ISE) for Linux: this is needed if you wish to build any of the hardware designs. This is non-free software, so it is not distributed with Linux.

Recommended versions: 6.2i is required for building some designs for Spartan-2E FPGAs. For other devices (Virtex-2, Virtex-4, Spartan-3), 8.1i is recommended. Subsequent versions are untested.

- ASL assembler - this is needed if you wish to rebuild any of the T80 control programs (e.g. the program shown in Figure 5.16). This software is not currently distributed with Debian Linux.

Recommended versions: 1.42 Beta.

- uDrawGraph - this is needed for viewing `.udg` graph files which are produced for debugging and visualisation purposes by some tools. uDrawGraph produces graphs like Figure 5.9. This is non-free software, so it is not distributed with Linux. It is only used for viewing results.

Recommended versions: 3.1.1.

- pygame - this Python module is needed to use the microprogramming GUI for MCGREP-1 (Figure 5.12). It is not needed for MCGREP-2. This software is available in Debian Linux: the package name is `python-pygame`.

Recommended versions: 1.7.1release.

- matplotlib - this Python module generates charts such as Figure 6.19. It is only used for viewing results. This software is available in Debian Linux: the package name is `python--matplotlib`.

Recommended versions: 0.90.1.

- Ghostscript - this is used to convert charts into PNG format for inclusion in the test report (section A.12). This software is available in Debian Linux: the package name is `gs-gpl`.

Recommended versions: 8.56.

- ImageMagick - this set of programs is useful for image manipulation. It can be used by the test software (section A.12) to generate visual image comparisons. This software is available in Debian Linux: the package name is `imagemagick`.

Recommended versions: 6.2.4.5.

If your computer is not able to execute x86 Linux programs directly, the easiest way to make use of the archive is likely to be through a virtual machine, PC emulator, or PC simulator. Such a program can be used to simulate the architectural environment of a 2008-era PC, allowing you to install an appropriate x86 version of Linux, perhaps from a CDROM `.iso` image file. All of the required programs are part of major Linux distributions such as Debian Etch and Slackware 12, which are available as free software. You will then be able to work with the archive programs in an environment that closely replicates the environment used to write them. If the software proves useful, it should be possible to recompile `gcc` and other programs for more modern hardware as the source code is included (section A.13).

A.3 Installing the Archive Software

The archive should be extracted to a directory with at least 1Gb of free space. The software is not intended to be shared between multiple users, and is not installed using the `root` system administrator account. Your home directory is a good place for it. Use the `tar` command to extract the archive:

```
tar xvjf jack-whitham-thesis-sw-dist-2008-xx-yy.tar.bz2
```

Next, `cd` to the newly created directory, and run the `install.sh` program. *Do not run this program as root.* The software does not install any programs outside of the directory created by `tar` and therefore does not require `root` privileges. (However, the software does create temporary files in `/tmp`.) The installation begins by displaying an information message. Press Enter, and the process will continue. Installation builds various programs and tests the environment on your computer, checking your version of Python and your C compiler. The correct completion message is as follows:

```
Install process complete.
Be sure to source "setup.sh" before you try to run any of the programs.
(See the manual for your shell to learn how to source a script.)
In the Bash shell, you can type the following to source setup.sh:
    . setup.sh
```

Installation will fail if one of the components listed in section A.2 is missing, or if the version you are using is different in some way to the version that was tested. After a successful installation:

1. Source the `setup.sh` file to load the correct environment for the software. In this context, “source” means that the commands in the file should be executed as if they were directly typed into the shell. This is not the same as running the script, because that will execute the commands within a child process. Files can be sourced using the `.` or `source` command in Bash.

The script changes the `PATH` variable, creates a new `MCGREP_PATH` variable, and runs a Python program to initialise the `/tmp` directory. `setup.sh`'s changes are not persistent: you must repeat this step every time you log in.

2. *Optional:* If you have the Xilinx ISE tools and you wish to build FPGA hardware designs, you should edit the `xilinx-ise-8.1.sh` script in the `xilinx-ise` subdirectory. This script, which contains an example, should load the Xilinx settings script for the version of Xilinx ISE you are using. If you want to build MCGREP-1 designs for the Spartan-2E FPGA, you must also edit the `xilinx-ise-6.2.sh` script. Version 6.2 of ISE appears to be required to build some Spartan-2E designs.
3. *Optional:* If you have an FPGA and wish to test hardware designs using the MCGREP-2 software, you should also edit the `download-bit.sh` script in the `xilinx-ise`. This script is executed with the absolute path to an FPGA bit file by tools such as `mcgrep-hardware_test.py`. It should program an FPGA with this bit file. The tools also expect the `serial-port` file to contain the Unix device name of a serial port that can be used to communicate with the FPGA, e.g. `/dev/ttyS0`. This may also need to be changed. See section A.11 for information about the required FPGA connections.

Once all files are installed, refer to sections A.4 through A.13 for information about the programs, libraries and hardware designs included within the appendix.

A.4 Building the MCGREP-1 Test Cases

To build the MCGREP-1 test cases, move to the test case directory (shown above). The `build` program offers the following options:

Post-installation Configuration Files

```
▷ /xilinx-ise
```

Xilinx ISE/FPGA configuration directory.

**MCGREP-1 Test Cases**

```
▷ /mcgrepl/testcases
```

Test case directory.

```
▷ /mcgrepl/testcases/build
```

Test case build program.

```
▷ /mcgrepl/testcases/bin
```

Test case output directory for binaries.



- `./build or`

Entering this command will build all the test cases for the MCGREP-1 platform. It automatically includes microcode, and patches the program binaries. The resulting programs are ready to run inside the MCGREP-1 simulator or on the hardware. These programs can be used to obtain the MCGREP-1 performance figures (section 5.3.6).

- `./build ror`

This command builds all the test cases for OpenRISC or MCGREP-1. Microcode is not included, and the programs only make use of operations supported by both OpenRISC and MCGREP-1. These programs can be used to obtain the OpenRISC performance figures (section 5.3.6).

- `./build aror`

This command builds the interference experiment (section 5.3.5). The binary to be used is:

```
/mcgrepl/testcases/bin/aes.bin
```

This should be executed on both the MCGREP-1 hardware and the OpenRISC CPU to obtain a full set of results.

Please note that it is not easy to extend the set of test cases because custom microprograms must be generated for each one using a manual process (Figure 5.12). To use the microprogramming GUI, you should use the `make_ucode.py` script in a test case subdirectory, but bear in mind that the inconvenience of this process was one of the main motivators for the automatic microprogram generator in MCGREP-2 (chapter 6).

MCGREP-1 Hardware

```

  ▷ /mcgrepl/hw
  Bitfile output directory.
  ▷ /mcgrepl/hw/build
  Hardware build program.
  ▷ /mcgrepl/hw/debug-monitor/mc_spartan2e.vhd
  MCGREP-1 VHDL file.
  ▷ /mcgrepl/hw/debug-monitor/mc_virtex2.vhd
  MCGREP-1 VHDL file.

```

**MCGREP-1 Simulator**

```

  ▷ /mcgrepl/testcases/*/run_orig.py
  Simulator program.
  ▷ /mcgrepl/testcases/*/run_accel.py
  Simulator program.

```



A.5 Using the MCGREP-1 Hardware Generator

To build the MCGREP-1 hardware, run the `build` program listed above. The program requires a working installation of Xilinx ISE. It builds four bitfiles:

1. `mcgrepl-eth-burchEd.bit` - MCGREP-1 plus test harness for “BurchEd B5” Spartan-2E board.
2. `mcgrepl-eth-virtex.bit` - MCGREP-1 plus test harness for “Amadeus” Virtex-2 board.
3. `openrisc-burchEd.bit` - OpenRISC OR1200 plus test harness for “BurchEd B5” Spartan-2E board.
4. `openrisc-virtex.bit` - OpenRISC OR1200 plus test harness for “Amadeus” Virtex-2 board.

Once the process has completed, you can find the MCGREP-1 VHDL source code in the locations shown above. This only interacts with external components via the test harness.

The bit files can be downloaded to an appropriate FPGA board for testing. However, the test harness assumes that the external interface will be compatible with the York RTS Group Virtual Lab (section A.11). If this interface is not available, you will need to change the top level VHDL files to implement your own external interface.

A.6 Using the MCGREP-1 Simulator

To use the MCGREP-1 simulator to run a test case, you should use either of the two simulator programs listed above, which can be found in the subdirectory of each test case. `run_orig.py` runs

MCGREP-2 Test Cases

```
▷ /testcases
```

Test case directory.

```
▷ /testcases/run_through.sh
```

Builds test cases with and without custom RFU configurations.

Uses checkpoints to compare the execution of each type of build.

```
▷ /testcases/or1kcompare.sh
```

Compares MCGREP-2 execution against the OpenRISC simulator.

**MCGREP-2 Hardware Generator**

```
▷ /mcgprep2-src/scripts/mcgprep_make_vhdl.py
```

Standalone VHDL generator.

```
▷ /mcgprep2-src/vhdl
```

Test system build directory.

```
▷ /mcgprep2-src/vhdl/build_ref_hw.sh
```

Builds the Avnet/Memec MM1 test system used for evaluation in chapter 6.



a test case in ORBIS32 mode only, without using any custom microprograms, while `run_accel.py` uses custom microprograms.

You will find that the MCGREP-2 simulator (sections 6.2.5 and A.9) is much faster than the MCGREP-1 simulator. However, the MCGREP-2 simulator only supports the OpenRISC ORBIS32 and Microblaze ISAs and the MCGREP-2 microprogramming interface. It cannot be used to run general MCGREP-1 programs because the microcode is not compatible.

A.7 Using the MCGREP-2 Test Cases

The MCGREP-2 test cases are built as part of an integrated experiment environment. They can also be built using the `mcgprep_testmaker.py` program: the `hskit.py` program includes examples of the usage of this program. The experiments are designed to execute unattended and produce results that are formatted into the tables and charts found in chapter 6.

A.8 Using the MCGREP-2 Hardware Generator

The MCGREP-2 hardware generator can be used in two ways:

- As part of the hardware building system for a supported FPGA, which generates the MCGREP-2 VHDL and then synthesises it. Currently, the supported FPGA for MCGREP-2 is the Spartan-3 `xc3s400-ft256-4` FPGA on the Avnet/Memec MM1 “mini module” FPGA board [22] (Figure A.2). The `build_ref_hw` program builds a bit file for this FPGA us-

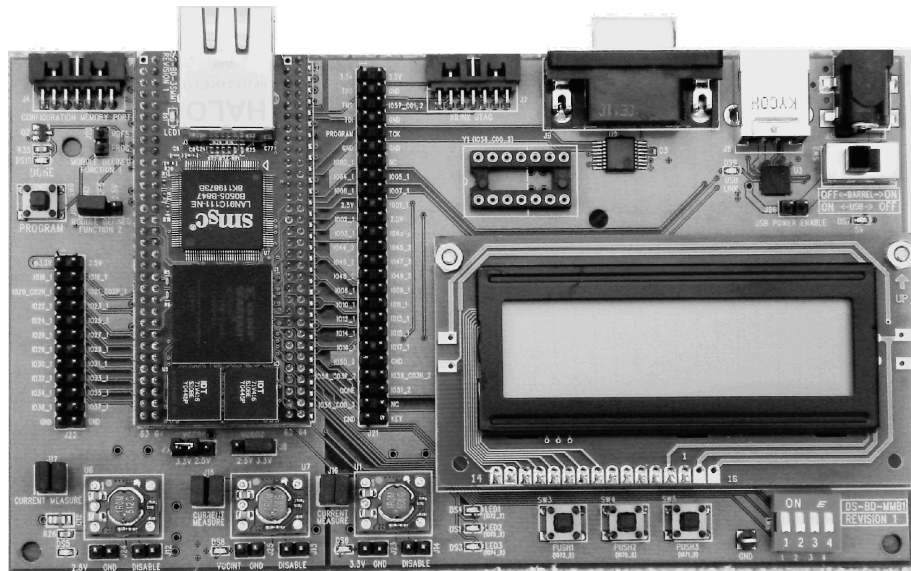


Figure A.2: Photograph of the MM1 “mini module” FPGA prototyping board, from Avnet/Memec documentation [22]. The mini module is plugged into the left-hand side of a development board that provides a serial port and JTAG interface in addition to a variety of other components (unused in this application) such as the display on the right-hand side.

ing Xilinx ISE. The bit file includes the test harness described in section 6.2.2.6. The `build_ref_core_hw` program builds the same system minus the test harness: in this configuration, the system is just an MCGREP-2 CPU plus memory and serial port drivers. Bit files are placed in `/mcgrep2-hw` along with temporary files produced by Xilinx ISE.

- As a standalone program. The `mcgrep_make_vhdl.py` program generates the self-contained VHDL source of an MCGREP-2 CPU. The program expects to be supplied with the name of a configuration file which specifies the parameters of the CPU to be generated. Sample configuration files can be found in `/mcgrep2-hw`: each has the extension `.cfg`. Some of the parameters that are supported are listed in Table A.1. Typically, this program is executed as follows:

```
mcgrep_make_vhdl.py -r mcgrep2-hw -n ref.cfg output.vhdl
```

This creates the file `output.vhdl` after reading the configuration file `ref.cfg` from the directory `mcgrep2-hw`.

The default bus used by MCGREP-2 CPUs is Wishbone [190]. However, the generator can also produce a version of the CPU with an On-chip Peripheral Bus (OPB) connector. If this is selected, then the output of the generator is a component for the Xilinx Embedded Development Kit (EDK) with an OPB interface. This component includes several files and is placed in a new subdirectory for inclusion in EDK, where it can act as a drop-in replacement for Microblaze if suitably configured (section D).

Parameter Name	Supported Values	Effect
num_units	Integer ≥ 1	Sets the total $l + m + n$ for the CPU (Figure 4.5).
hw_full_debug_chain	Boolean	Synthesise a long debugging chain that includes all CPU components rather than a subset.
memory_latency	≥ 1	Expect the specified memory latency. (Note: does <i>not</i> affect the latency of RAM accesses in the simulator - a command-line parameter is used for that purpose.)
multiply_on_any_unit	Boolean	Add a multiplier to every functional unit rather than just the first.
target	"spartan2e", "spartan3"	Selects the block RAM driver to be used. The "spartan3" selection is also suitable for Virtex-II and subsequent devices.
arch_name	"openrisc", "microblaze"	Selects the ISA of the CPU.

Table A.1: Configuration parameters for the MCGREP-2 hardware generator, tester, and simulator.

MCGREP-2 Simulator

▷ /mcgprep2-src/scripts/mcgprep_simulator.py

Standalone simulator.

▷ /mcgprep2-src/scripts/mcgprep_mcuc_test.py

Debugging aid for microprogramming problems.

▷ /mcgprep2-src/scripts/mcgprep_hardware_test.py

Compares hardware and simulator execution using debugging mechanism.



A.9 Using the MCGREP-2 Simulator

The MCGREP-2 simulator is widely used by test cases and experiments. For example, it is used to check the correct operation of every test case in `/testcases`, and it is also used to carry out the experiments in chapters 6 to 8. Many programs extend the simulator with hooks (section 6.2.5), so the simulator is executed via Python. However, the simulator can also be executed manually by running `mcgrep_simulator.py`.

The simulator program uses a configuration file like the one accepted by the hardware generator. (The same file can be used by both.) The supported parameters are listed in Table A.1. Microprograms are not portable between different configurations of MCGREP-2, so it is important to ensure that the simulator, the hardware and the code generator all share the same configuration.

The simulator's debugging switch (`-d`) causes traces to be emitted at the microprogram level. This is useful for debugging microprograms. More sophisticated debugging is possible using `mcgrep_mcuc_test.py`, an enhanced version of the simulator which supports the following additional features:

- `--debug-entry`: activates debugging after a specific microprogram state has been reached. This avoids the need to trace all the microinstructions executed before that point.
- `--dump_to`: dumps memory and registers to two files beginning with the specified name when the simulator exits.
- `--rebuild-debugging`: keep each successive version of the automatically generated C components of the simulator.

Adding features to MCGREP-2 CPUs may involve writing both VHDL and C implementations for each new type of microoperation. This type of extension can be debugged using `mcgrep_hardware_test.py`, which compares the functionality of the hardware and software models (section 6.4.1). This is done through the debugging harness on the embedded system (implemented by `System_Harness.vhd`), which communicates by a RS232 serial link with the `hardware_test.py` program on your workstation. Section A.11 has information about setting up an FPGA prototyping board to host the embedded system.

A.10 Extending MCGREP-2

The bulk of the MCGREP-2 software is stored within the `/mcgrep2-src/mcgrep` directory. This directory is a Python package named `mcgrep`: if changes are made, it may be necessary to run `install.sh` again to update Python.

The parts of MCGREP-2 that are explicitly extensible are found in `/mcgrep2-src/mcgrep/architectures`. The `microblaze` architecture is derived from the base `openrisc` architecture. Many new features can be added by deriving a new architecture from one of these. Each directory must include a standard set of files which are loaded by the `arch` module of MCGREP-2. The architecture to be used is selected by a configuration parameter (Table A.1).

The trace generating software (section 6.3) is not part of MCGREP-2: it communicates with the MCGREP-2 tools via an interface as shown in Figure 6.7. This software can be found in `/tracegen`. The trace generator itself is found in `sssopt.c`, with the MCGREP-2 interface in `mcgrepinf.c` and the machine code interface in `isadecode.c`.

The WCET reduction software (section 7.2) is located in `/wctreduce/rtas.py`. This code exposes a variety of interfaces to allow extensions and experiments: usage examples can be found in `/wctreduce/shorttest.py` and `/wctreduce/control.py`.

A.11 Connecting an FPGA

Some of the tests and experiments require an FPGA. At present, FPGAs come in a wide variety of packages on a wide variety of prototyping boards. Although VHDL is standard [17], there is no universal standard for FPGA hardware or prototyping boards, and each device has different pin connections and peripherals. Therefore, the components of the work that require an FPGA will probably need to be changed to meet your requirements.

MCGREP-1 experiments make use of the York RTS Group Virtual Lab. This provides a terminal interface to FPGA hardware, allowing users to send commands to FPGA hardware and see the results in an applet (Figure 5.16). From the perspective of the T80 CPU that acts as a microcontroller for the debugging hardware (section 5.3.1), the 40x25 text display and the serial input are memory mapped devices. If necessary, the Virtual Lab features can be recreated by replacing these two devices with the implementations in `mcgrep2-src/vhdl/src/generic/vga_module.vhd` and `mcgrep2-src/vhdl/src/generic/uart_module.vhd`. This can be done by changing the `mcgrep1/hw/common/monitor_bus_bridge.vhd` module so that bus transactions are sent to local devices rather than being encoded for the external Virtual Lab interface.

Because of the potential difficulty of reproducing Virtual Lab features in future systems, the MCGREP-2 experiments and tests use a simple RS232 serial connection to communicate with an FPGA. The FPGA prototyping board used for the MCGREP-2 tests is the Avnet/Memec MM1 “mini module” (section A.8), but this can be replaced by any prototyping board that provides RS232 line drivers and at least 1Mb of SRAM in addition to a suitable FPGA. To use a different FPGA or prototyping board with the existing tools, refer to the files in `mcgrep2-src/vhdl/boards/ref`. These specify the prototyping board parameters. `ref.vhd` is the top level VHDL file and should provide pins for a RS232 serial connection, a clock and SRAM. `ref.ucf` specifies the pin names. The other files are used by various parts of the Xilinx synthesis process.

The `ref` subdirectory can be copied to create an entirely new board target. The `build.ref_hw` script in `mcgrep2-src/vhdl` should be modified to specify the board directory (`NAME`) and the FPGA type (`PART`). Various other board targets exist in the archive, but not all of these have been tested.

The MCGREP-2 software does not set the baud rate of the serial connection on the workstation, so a terminal emulator program must be used to do this. The baud rate expected by the MM1 system is 57600 bits per second (8 bits per character, no parity, 1 stop bit, no flow control). This can be changed by modifying the `uart_divisor` parameter of the `System` component in `ref.vhd`. The equation is:

$$d = \frac{f}{64b} \tag{A.1}$$

where d is the divisor to be rounded to nearest integer, f is the FPGA input clock frequency in Hz, and b is the baud rate in bits per second. For the MM1 system, $f = 100\text{MHz}$ and the divisor $d = 27$.

The test programs call the `download-bit.sh` script in `xilinx-ise` to send bit files to the FPGA. This can call any other program to do the work: the Xilinx Impact program is a possibility.

Appendix Software Tests

- ▷ /utils/test
Test case directory.
- ▷ /results
Test results directory.
- ▷ /regtest
Regression test data.



By default, the script uses the Virtual Lab to download bit files.

A.12 Appendix Software Tests

The tests produce most of the results printed in this thesis. Some tests require the Xilinx tools, others require an FPGA. The `b` programs in the test case directory have the following functions:

- `b1.py`: installation sanity checks. This test is very short.
- `b2.py`: synthesis tests. All of the tests carried out by this program require Xilinx ISE. This test takes around 3 hours on a 2008-era PC.
- `b3.py`: software tests, part 1. MCGREP-1 software is tested, along with some of MCGREP-2. This test takes around 24 hours.
- `b3a.py`: software tests, part 2. MCGREP-2 tests are completed, and the WCET reduction experiments described in chapters 7 and 8 are performed. This test takes around 24 hours.
- `b4.py`: hardware test. The MCGREP-2 hardware is compared against the simulator in both machine code and custom RFU execution mode for each benchmark. In this test case, the trace generator software is executed on the FPGA itself before each benchmark is executed. Before starting this test, see section A.11. This test takes around 4 hours.
- `b6.py`: result production. The test results are finalised and a report is produced in `/results/reportcharts.html`. This report can be viewed using a Web browser. If the ImageMagick program `compare` is installed, the generated script `/results/reportcharts.sh` can be executed to generate a visual comparison between the regression test data and the latest results. This test is very short.

A.13 Third Party Software and Hardware

The archive includes a number of software programs and hardware designs written by others. These are redistributed under the terms of the GNU General Public License version 2. This licence can be found in the file named `/LICENSE` in the root of the archive.

Complete source code for all of the following programs can be found in `/3rdparty-sw/src`:

- `/3rdparty-sw/glpk`: the GNU Linear Programming Kit, version 4.22 [100]. This component is sourced from the Free Software Foundation website. No modifications have been made.
- `/3rdparty-sw/openrisctools`: `newlib`, `gcc` and `binutils` for the OpenRISC ORBIS32 ISA [151]. These components are sourced from the Opencores.org website. Some changes were necessary in order to compile this software: these are present within three patch files in the `src` directory.
- `/3rdparty-sw/microblazetools`: `newlib`, `gcc`, `libgloss` and `binutils` for the Microblaze ISA [286]. These components are sourced from Petalogix. Some changes were necessary in order to compile this software: these are present within one patch files in the `src` directory.
- `/3rdparty-sw/openrisctools`: `libgloss` for the OpenRISC ORBIS32 ISA. This component is based on `libgloss` for Microblaze. It includes modifications to use the simulator system call interface (section 6.4.1) for access to files on the host workstation.
- `/3rdparty-sw/orlksim`: the simulator for ORBIS32. This component is sourced from the Opencores.org website. No modifications have been made.
- `hex2rom`: this converts binary files into read-only memory implemented in block RAM. It has been modified to generate ROMs with a Wishbone bus.

Complete VHDL/Verilog source code for the following third-party hardware designs can be found in `/3rdparty-cores`:

- `/3rdparty-cores/t80`: the T80 CPU [269]. This component is sourced from the Opencores.org website.
- `/3rdparty-cores/or1200`: the OpenRISC CPU [151]. This component is sourced from the Opencores.org website.
- `/3rdparty-cores/mem_ctrl`: a memory controller from Opencores.org.

Some of the benchmark programs are *not* licenced under the GNU General Public Licence version 2. In these cases, a file named `LICENSE` is present in the benchmark source directory detailing the terms for that specific program. These terms only apply to files in that directory, and have previously permitted redistribution of the code within MIBench [108] and Mediabench [154].

Finally, the following additional third-party code is used:

- `/utils/virtual-python.py`: clones an installation of Python. This component is public domain software.

Appendix B

Experimental Data and Test Examples

```
_III_huffdecode:
...
*   ac48:   b8 86 00 58   l.srli r4,r6,0x18
      ac4c:   a5 84 00 0f   l.andi r12,r4,0xf
      ac50:   bc 2c 00 00   l.sfnei r12,0x0
      ac54:   10 00 00 0b   l.bf ac80 <_III_huffdecode+0x844>
      ac58:   9f a0 00 01   l.addi r29,r0,0x1
*   ac5c:   9d 40 00 00   l.addi r10,r0,0x0
      ac60:   d4 10 50 00   l.sw 0x0(r16),r10
*   ac64:   ba a6 00 54   l.srli r21,r6,0x14
      ac68:   a5 95 00 0f   l.andi r12,r21,0xf
      ac6c:   bc 2c 00 00   l.sfnei r12,0x0
      ac70:   13 ff ff bc   l.bf ab60 <_III_huffdecode+0x724>
      ac74:   9c a0 00 01   l.addi r5,r0,0x1
*   ac78:   03 ff ff ce   l.j abb0 <_III_huffdecode+0x774>
      ac7c:   9e e0 00 00   l.addi r23,r0,0x0
*   ac80:   84 62 fe fc   l.lwz r3,0xfffffec(r2)
      ac84:   e0 9d 60 08   l.sll r4,r29,r12
      ac88:   e3 63 20 03   l.and r27,r3,r4
      ac8c:   bc 1b 00 00   l.sfeqi r27,0x0
      ac90:   10 00 00 0e   l.bf acc8 <_III_huffdecode+0x88c>
      ac94:   b8 ec 00 02   l.slli r7,r12,0x2
*   ac98:   e3 e7 10 00   l.add r31,r7,r2
      ac9c:   85 7f ff 24   l.lwz r11,0xffffffff24(r31)
      aca0:   9e 6e ff ff   l.addi r19,r14,0xffffffff
      aca4:   9e 20 00 01   l.addi r17,r0,0x1
      aca8:   e1 f1 98 08   l.sll r15,r17,r19
      acac:   e1 92 78 03   l.and r12,r18,r15
      acb0:   bc 0c 00 00   l.sfeqi r12,0x0
      acb4:   10 00 00 03   l.bf acc0 <_III_huffdecode+0x884>
      acb8:   a9 d3 00 00   l.ori r14,r19,0x0
*   acbc:   e1 60 58 02   l.sub r11,r0,r11
*   acc0:   03 ff ff e9   l.j ac64 <_III_huffdecode+0x828>
      acc4:   d4 10 58 00   l.sw 0x0(r16),r11
...
```

Figure B.1: Hotspot example: a fragment of code from the `mad` benchmark program [154]. Each `*` symbol marks the start of a new basic block.

p	MCGREP mc only	MCGREP mc+μc	OpenRISC cache	OpenRISC no cache
4000	$y = 4.261e + 09$ $n = 1377896$	$y = 1.644e + 09$ $n = 528937$	$y = 1.027e + 09$ $n = 308939$	$y = 3.647e + 09$ $n = 1060987$
8000	$y = 4.235e + 09$ $n = 597611$	$y = 1.634e + 09$ $n = 230031$	$y = 8.866e + 08$ $n = 121077$	$y = 3.578e + 09$ $n = 481262$
12000	$y = 4.227e + 09$ $n = 381325$	$y = 1.631e + 09$ $n = 146791$	$y = 8.493e + 08$ $n = 75015$	$y = 3.558e + 09$ $n = 311019$
16000	$y = 4.224e + 09$ $n = 279918$	$y = 1.629e + 09$ $n = 107898$	$y = 8.197e + 08$ $n = 53489$	$y = 3.548e + 09$ $n = 229803$
20000	$y = 4.222e + 09$ $n = 221145$	$y = 1.629e + 09$ $n = 85252$	$y = 8.020e + 08$ $n = 41507$	$y = 3.542e + 09$ $n = 182199$
24000	$y = 4.221e + 09$ $n = 182822$	$y = 1.628e + 09$ $n = 70459$	$y = 7.906e + 08$ $n = 33895$	$y = 3.538e + 09$ $n = 150945$
28000	$y = 4.220e + 09$ $n = 155772$	$y = 1.628e + 09$ $n = 60033$	$y = 7.827e + 08$ $n = 28645$	$y = 3.536e + 09$ $n = 128820$
32000	$y = 4.219e + 09$ $n = 135696$	$y = 1.628e + 09$ $n = 52312$	$y = 7.768e + 08$ $n = 24798$	$y = 3.534e + 09$ $n = 112370$
36000	$y = 4.218e + 09$ $n = 120239$	$y = 1.627e + 09$ $n = 46357$	$y = 7.722e + 08$ $n = 21862$	$y = 3.532e + 09$ $n = 99664$
40000	$y = 4.218e + 09$ $n = 107913$	$y = 1.627e + 09$ $n = 41610$	$y = 7.686e + 08$ $n = 19544$	$y = 3.531e + 09$ $n = 89542$
44000	$y = 4.218e + 09$ $n = 97880$	$y = 1.627e + 09$ $n = 37735$	$y = 7.656e + 08$ $n = 17670$	$y = 3.530e + 09$ $n = 81267$
48000	$y = 4.217e + 09$ $n = 89567$	$y = 1.627e + 09$ $n = 34539$	$y = 7.631e + 08$ $n = 16124$	$y = 3.529e + 09$ $n = 74398$
52000	$y = 4.217e + 09$ $n = 82548$	$y = 1.627e + 09$ $n = 31831$	$y = 7.610e + 08$ $n = 14827$	$y = 3.529e + 09$ $n = 68602$
56000	$y = 4.217e + 09$ $n = 76546$	$y = 1.627e + 09$ $n = 29519$	$y = 7.592e + 08$ $n = 13723$	$y = 3.528e + 09$ $n = 63636$
∞	$y = 4.214e + 09$ $n = 1$	$y = 1.626e + 09$ $n = 1$	$y = 7.350e + 08$ $n = 1$	$y = 3.521e + 09$ $n = 1$

Table B.1: MCGREP-1 CPU: Results of interference experiment (raw data). Each table cell indicates the values of y (clock cycles) and n (no units) measured on each platform with each value of p . See also Figure 5.21.

Benchmark	MCGREP mc only	OpenRISC no cache	Microblaze no cache	MCGREP mc+μc
aes	1.08e+07	7.63e+06	7.08e+06	6.88e+06
crc32	1.77e+07	1.38e+07	1.38e+07	7.23e+06
dijkstra	9.33e+08	7.73e+08	6.19e+08	6.08e+08
g721	1.03e+09	8.86e+08	9.35e+08	7.91e+08
jpeg	3.64e+07	2.87e+07	2.24e+07	2.55e+07
mad	1.65e+08	1.25e+08	1.35e+08	1.12e+08
qsort	2.26e+07	2.37e+07	1.87e+07	1.72e+07
sha	2.35e+08	1.93e+08	1.93e+08	1.06e+08

Table B.2: MCGREP-1 CPU: Benchmark execution times on four platforms (raw data, figures in clock cycles). See also Figure 5.22.


```

2270: e0 64 48 00  l.add r3,r4,r9
2274: 9c c3 00 04  l.addi r6,r3,0x4
2278: 84 a3 00 00  l.lwz r5,0x0(r3)
227c: 84 86 00 00  l.lwz r4,0x0(r6)
2280: e5 a5 20 00  l.sfles r5,r4
2284: 10 00 00 05  l.bf 2298 <_Bubble_Sort+0x54>
2288: 9c e7 00 01  l.addi r7,r7,0x1
228c: d4 03 20 00  l.sw 0x0(r3),r4
2290: 9d 00 00 01  l.addi r8,r0,0x1
2294: d4 06 28 00  l.sw 0x0(r6),r5
2298: bd a7 00 62  l.sflesi r7,0x62
229c: 13 ff ff f5  l.bf 2270 <_Bubble_Sort+0x2c>
22a0: b8 87 00 02  l.slli r4,r7,0x2
22a4: 15 00 00 00  l.nop 0

```

Figure B.2: Bubble sort inner loop, as ORBIS32 assembly. C source appears in Figure 7.7.

Program	Traces	Microprogram Space Used	Untraced E.T.	Traced E.T.	Normalised
aes	1	383	5.33e+06	2.74e+06	0.515
bitcount	3	354	1.71e+08	1.53e+08	0.898
blowfish	2	406	7.14e+07	3.97e+07	0.557
crc32	2	423	9.27e+06	3.60e+06	0.389
dijkstra	1	384	5.05e+08	2.92e+08	0.578
g721	1	376	5.96e+08	4.82e+08	0.809
gsm	1	347	7.79e+07	5.83e+07	0.749
jpeg	2	352	1.47e+08	1.06e+08	0.721
mad	3	518	8.63e+07	7.84e+07	0.909
patricia	2	355	1.94e+08	1.46e+08	0.756
qsort	1	366	1.65e+07	1.59e+07	0.966
sha	4	280	1.24e+08	8.33e+07	0.674
stringsearch	3	414	1.09e+06	9.78e+05	0.894

Table B.3: MCGREP-2: Benchmark execution times on two platforms (raw data: execution time figures are given in clock cycles). See also Figure 6.19.

Block Size	Time taken for					Normalised
	JPEG	AES	Blowfish	Upload	Total	
m/c only	1.04e+08	2.34e+07	2.47e+07	0.00e+00	1.52e+08	1.000
32768	8.17e+07	1.19e+07	1.03e+07	1.33e+06	1.05e+08	1.449
16384	8.17e+07	1.19e+07	1.03e+07	2.40e+06	1.06e+08	1.435
8192	8.17e+07	1.19e+07	1.03e+07	4.81e+06	1.09e+08	1.403
4096	8.17e+07	1.19e+07	1.04e+07	9.34e+06	1.13e+08	1.347
2048	8.17e+07	1.19e+07	1.04e+07	1.84e+07	1.22e+08	1.246
1024	8.17e+07	1.19e+07	1.04e+07	3.68e+07	1.41e+08	1.083
512	8.17e+07	1.19e+07	1.04e+07	7.37e+07	1.78e+08	0.858

Table B.4: Timings for jab executions with different block sizes (see section 6.4.10).

Program	8192 bits 2 units	16384 bits 2 units	24576 bits 3 units	32768 bits 3 units	65536 bits 3 units
bs	4.9	6.0	6.6	6.6	6.6
bubble	17.7	52.0	62.0	66.7	67.2
cnt	35.9	53.9	64.2	68.0	75.9
compress	8.3	14.5	22.2	35.4	49.0
crc	7.4	15.4	19.0	20.6	21.7
div	15.4	33.1	38.6	40.2	43.7
duff	5.4	36.1	36.1	56.4	59.3
edn	26.7	55.4	65.9	75.2	94.2
expint	45.7	64.4	74.7	77.1	93.9
fdct		39.5	43.8	43.8	149.0
fibcall	8.4	9.0	9.0	9.0	9.0
fir	14.3	30.3	35.3	42.5	48.4
insertsort	19.4	55.1	55.1	58.5	67.1
janne_complex	5.7	6.6	8.8	8.8	8.8
jfdctint	27.5	52.0	61.7	65.4	73.5
matmult	25.5	45.2	52.4	61.7	77.5
ndes	1.5	4.8	5.6	8.9	22.8
ns	47.5	60.9	60.9	62.7	63.9

Table B.5: Improvements made by a hybrid (scratchpad and RFU) architecture versus an instruction scratchpad architecture. Each value is a percentage. For each architecture and program, the value is equal to $100 \frac{i}{h}$, where h is the hybrid WCET, and i is the instruction scratchpad WCET. See also Figure 8.3.

Appendix C

OpenRISC ORBIS32 Quick Reference

This section lists the OpenRISC ORBIS32 ISA subset used for MCGREP-1 and MCGREP-2. For the complete ISA definition, refer to [151].

Instruction	Instruction Word					Semantics
	[31:26]	[25:21]	[20:16]	[15:11]	[10:0]	
ladd rD,rA,rB	111000	rD	rA	rB	00000000000	rD := rA + rB
laddi rD,rA,Imm	100111	rD	rA		Imm	rD := rA + Imm
land rD,rA,rB	111000	rD	rA	rB	00000000000	rD := rA & rB
landi rD,rA,Imm	101001	rD	rA		Imm	rD := rA & Imm
lbf Imm	000100	Imm				<i>Branch if flag:</i> pc := pc + Imm
lbnf Imm	000011	Imm				<i>Branch if not flag:</i> pc := pc + Imm
lj Imm	000000	Imm				pc := pc + Imm
ljalr rD	010010	0000000000		rB	00000000000	r9 := pc pc := rB
ljr rD	010001	0000000000		rB	00000000000	pc := rB
llbs rD, Imm(rA)	100100	rD	rA	Imm		<i>Load 1 byte (sign ex.):</i> rD := [rA + Imm]
llbz rD, Imm(rA)	100011	rD	rA	Imm		<i>Load 1 byte:</i> rD := [rA + Imm]
llhs rD, Imm(rA)	100110	rD	rA	Imm		<i>Load 1 half-word (sign ex.):</i> rD := [rA + Imm]
llhz rD, Imm(rA)	100101	rD	rA	Imm		<i>Load 1 half-word:</i> rD := [rA + Imm]
llws rD, Imm(rA)	100010	rD	rA	Imm		<i>Load 1 word (sign ex.):</i> rD := [rA + Imm]
llwz rD, Imm(rA)	100001	rD	rA	Imm		<i>Load 1 word:</i> rD := [rA + Imm]
lmovhi rD,Imm	000110	rD	Imm			rD := Imm << 16

Instruction	Instruction Word					Semantics
	[31:26]	[25:21]	[20:16]	[15:11]	[10:0]	
lmul rD,rA,rB	111000	rD	rA	rB	000000000000	rD := rA × rB
lmuli rD,rA,Imm	101100	rD	rA	Imm		rD := rA × Imm
lnop	000101	01000	000000000000000000000000			<i>No operation</i>
lor rD,rA,rB	111000	rD	rA	rB	000000000000	rD := rA rB
lori rD,rA,Imm	101010	rD	rA	Imm		rD := rA Imm
lsb Imm(rA), rB	110110	Imm[15:11]	rA	rB	Imm [10:0]	<i>Store 1 byte:</i> [rA + Imm] := rB
lsfeq rA,rB	111001	00000	rA	rB	000000000000	flag := rA = rB
lsfeqi rA,Imm	101111	00000	rA	Imm		flag := rA = Imm
lsfges rA,rB	111001	01011	rA	rB	000000000000	flag := rA ≥ rB (signed)
lsfgesi rA,Imm	101111	01011	rA	Imm		flag := rA ≥ Imm (signed)
lsfgeu rA,rB	111001	00011	rA	rB	000000000000	flag := rA ≥ rB
lsfgeui rA,Imm	101111	00011	rA	Imm		flag := rA ≥ Imm
lsfgts rA,rB	111001	01010	rA	rB	000000000000	flag := rA > rB (signed)
lsfgtsi rA,Imm	101111	01010	rA	Imm		flag := rA > Imm (signed)
lsfgtu rA,rB	111001	00010	rA	rB	000000000000	flag := rA > rB
lsfgtui rA,Imm	101111	00010	rA	Imm		flag := rA > Imm
lsfles rA,rB	111001	01101	rA	rB	000000000000	flag := rB ≥ rA (signed)
lsflesi rA,Imm	101111	01101	rA	Imm		flag := Imm ≥ rA (signed)
lsfleu rA,rB	111001	00101	rA	rB	000000000000	flag := rB ≥ rA
lsfleui rA,Imm	101111	00101	rA	Imm		flag := Imm ≥ rA
lsfits rA,rB	111001	01100	rA	rB	000000000000	flag := rB > rA (signed)
lsfitsi rA,Imm	101111	01100	rA	Imm		flag := Imm > rA (signed)
lsftu rA,rB	111001	00100	rA	rB	000000000000	flag := rB > rA
lsftui rA,Imm	101111	00100	rA	Imm		flag := Imm > rA
lsfne rA,rB	111001	00001	rA	rB	000000000000	flag := rA ≠ rB
lsfnei rA,Imm	101111	00001	rA	Imm		flag := rA ≠ Imm
lsh Imm(rA), rB	110111	Imm[15:11]	rA	rB	Imm [10:0]	<i>Store 1 half-word:</i> [rA + Imm] := rB
lll rD,rA,rB	111000	rD	rA	rB	000000000000	rD := rA << rB

Instruction	Instruction Word					Semantics
	[31:26]	[25:21]	[20:16]	[15:11]	[10:0]	
l.slli rD,rA,Imm	101110	rD	rA		Imm	rD := rA << Imm
l.sra rD,rA,rB	111000	rD	rA	rB	00000000000	rD := rA >> rB (signed)
l.srai rD,rA,Imm	101110	rD	rA		Imm	rD := rA >> Imm
l.srl rD,rA,rB	111000	rD	rA	rB	00000000000	rD := rA >> rB
l.srli rD,rA,Imm	101110	rD	rA		Imm	rD := rA >> Imm
l.sub rD,rA,rB	111000	rD	rA	rB	00000000000	rD := rA - rB
l.sw Imm(rA), rB	110101	Imm[15:11]	rA	rB	Imm [10:0]	<i>Store 1 word:</i> [rA + Imm] := rB
l.xor rD,rA,rB	111000	rD	rA	rB	00000000000	rD := rA xor rB
l.xori rD,rA,Imm	101011	rD	rA		Imm	rD := rA xor Imm

Appendix D

Microblaze Extensions for MCGREP-2

All of the experiments described in chapters 6 through 8 make use of the OpenRISC ORBIS32 ISA, but other ISAs could be used following appropriate modifications to MCGREP-2, the trace generator, and the experimental software. Both MCGREP-2 and the trace generator can be extended to support new ISAs through extensions to a MCGREP-2 architecture definition (section 6.2.1). New definitions can specify new instruction sets, extended ALU functions, and additional supporting hardware. A new definition for the Microblaze ISA [286] is motivated by the following issues:

- **Improved Compilation:**

It has been found that the OpenRISC C compiler does not generate code as efficiently as the Microblaze compiler (section 5.3.7). This is likely to be a combination of the effects of the C compiler design and the ISA design. Since improving the compiler design is outside the scope of this thesis, this motivates support for a new ISA.

- **OS/RTOS Support:**

OpenRISC can run programs of unlimited complexity, and can be used as the CPU for Linux-based embedded systems, but Linux support requires an MMU, which is not provided by MCGREP-2. uClinux or an RTOS such as RTEMS would be a better choice of OS for MCGREP-2, because these do not require an MMU. However, support for OpenRISC is deprecated in both RTEMS and uClinux. On the other hand, Microblaze ports of uClinux currently exist, and several RTOSs have also been ported to Microblaze.

- **Xilinx EDK Support:**

MCGREP-2 CPUs are normally generated as a VHDL component, where they can be manually built into a hardware design. However, recent trends in embedded systems design have been moving towards system-level design tools that allow a complete device to be built from IP cores without HDL coding. The Xilinx Embedded Development Kit (EDK) is one such tool. EDK allows systems to be constructed in block form from reusable IP, such as CPUs, memory controllers and other hardware drivers. Microblaze is one IP core that can be used in this fashion. MCGREP-2 support for EDK component generation would allow MCGREP-2 CPUs to be used as a plug-in replacement for Microblaze within EDK designs, allowing embedded systems designers to make use of MCGREP-2 CPUs without needing to work with VHDL code.

This appendix describes the process of extending the MCGREP-2 OpenRISC definition to support the Microblaze ISA, with the goals of (a) carrying the experiments described by chapter 6 with Mi-

Instruction	Carry In	Carry Out	B source	rD Output
add	Zero	Yes	Register	$rA + rB$
addk	Zero	No	Register	$rA + rB$
addc	Yes	Yes	Register	$rA + rB + C$
addkc	Yes	No	Register	$rA + rB + C$
rsub	Zero	Yes	Register	$rB - rA$
rsubk	Zero	No	Register	$rB - rA$
rsubc	Yes	Yes	Register	$rB - rA + C$
rsubkc	Yes	No	Register	$rB - rA + C$
addi	Zero	Yes	Immediate	$rA + I$
addik	Zero	No	Immediate	$rA + I$
addic	Yes	Yes	Immediate	$rA + I + C$
addikc	Yes	No	Immediate	$rA + I + C$
rsubi	Zero	Yes	Immediate	$rB - rA$
rsubik	Zero	No	Immediate	$rB - rA$
rsubic	Yes	Yes	Immediate	$rB - rA + C$
rsubikc	Yes	No	Immediate	$rB - rA + C$

Table D.1: Microblaze includes many different addition and subtraction instructions with different carry and borrow behaviour [286].

croblaze substituted for ORBIS32, and (b) booting uClinux on MCGREP-2. The implementation is based on the Microblaze ISA manual [286], the Microblaze `gcc` source, and test programs executed on a Microblaze CPU.

Sections D.1 to D.4 describe parts of the Microblaze architecture description. Section D.5 describes repetitions of the tests from section 6.4 with the alternate ISA, and section D.6 describes the process of booting uClinux on MCGREP-2 CPUs.

D.1 ALU Features

The low-level features of the Microblaze ALU are not the same as those of the OpenRISC ALU. One difference is that reverse subtraction is used ($B - A$ instead of $A - B$). Another is the lack of comparison operations: instead of using one of a large number of “set flag” operations to compare values, Microblaze programs carry out comparisons as an implicit part of branch operations. There are two explicit comparison operations for the operations that cannot be implemented in this way.

A third difference is a change to the way that carry flags are supported. Addition and subtraction operations have eight subtypes depending on the input source and the on carry effects (Table D.1). Only some of these have equivalents in ORBIS32, and none of the “add with carry” operations in ORBIS32 are generated by `gcc`. Therefore, support for the Microblaze ISA introduces carry support to MCGREP-2 CPUs. Finally, some of the Microblaze operations provide features that are not available in the OpenRISC ISA. For one example, the “shift right with carry” operation shifts the carry flag into the most significant bit of a register. Two other examples include the sign extension operations and the “and not” instructions.

The large number of possible ALU operations would waste microprogram space if one microinstruction was allocated to each operation, since most differ only in their settings of the ALU control lines. (The same is true for ORBIS32, but fewer distinct operations are used in that case.)

Instruction	Handled Using
andn	xor followed by and.
idiv/idivu	Division microprogram: uses some dedicated division hardware. ALU is used as adder or subtractor depending on division state.
sext8/sext16	Left shift followed by arithmetic right shift.
sra/src/srl	Right shift.

Table D.2: Implementation of irregular Microblaze instructions for MCGREP-2. Simpler ALU functions are reused to implement these operations wherever possible.

Therefore, a special `ALU_FROM_IWORD` setting is introduced to allow the ALU function to be programmed directly from the instruction word. This allows most ALU operations to share a single microinstruction.

The ALU operations that do not fit into this microinstruction could be thought of as “irregular” and are handled as special cases by short microprograms. On a genuine Microblaze, it is likely that all of the irregular operations do not need to be handled as special cases: their function is probably easy to support within the ALU as a side effect of other functions. (For instance, “and not” is related to subtraction, with “and” in place of addition.) But on MCGREP-2 CPUs, it is easiest to support these operations using microprograms. The irregular operations are listed in Table D.2.

D.2 ISA and Built-in Microprogram

An entirely new interpreter microprogram is required to support the Microblaze ISA, but the instruction decoder and microcode execution components are reused. All of the new code can be found in `interpreter.py` in the ISA architecture definition directory. Much of the code is very similar to its analogue for OpenRISC, with appropriate changes for the Microblaze ISA.

It is necessary to support division operations in the microprogram in order to boot uClinux for the Microblaze ISA without requiring recompilation. The operation is partly implemented by hardware since it involves two simultaneous activities: shifting a quotient register, and adding or subtracting values in a result register. Communication between the division hardware and the microprogram takes place via a special ALU control code and the LSU interface, which can be used since the division hardware is only present on the first functional unit.

Another change is introduced for conditional branch operations, which evaluate a comparison. To support these operations, the comparison subsystem is moved out of the ALU into a second comparator. The type of comparison to be carried out may be set by either the instruction word or by the microprogram.

D.3 Interrupt and Status Register Support

An OS such as uClinux requires interrupt support to support time-sliced multitasking and allow efficient device accesses. This is present in MCGREP-1, but not the ORBIS32 architecture for MCGREP-2, because interrupt support is not needed for any of the experiments in chapters 6 through 8. Interrupt support must be reintroduced for the Microblaze ISA version to be usable with

an OS. Additionally, the status registers used to control the interrupt process must be implemented along with the instructions used to access them: `msrset`, `msrclr`, `mfs`, and `mts`.

On MCGREP-2, interrupts can be supported within the microprogram control unit. On interrupt, execution is rerouted from machine code to an interrupt microprogram, which saves the old program counter and jumps to the fixed address `0x10`. Conventionally [286], this address contains a jump instruction pointing at the OS *interrupt service routine* (ISR). This process is inhibited by disabling interrupts in a status register. It is also temporarily inhibited when instruction execution must be atomic, which is used for delayed branches and after immediate load operations (`imm`). (Inhibiting interrupts in these locations makes it easier to resume execution after handling.) Some additional hardware is needed to control the uPC and implement the two status registers of interest: MSR (the machine status register) and ESR (the exception status register).

D.4 Trace-style Scheduler

The trace generator described in section 6.3 may be modified for Microblaze by changing the front end (section 6.3.2), which decodes the input program, and the output stage (section 6.3.5), which generates the microprogram. No changes are required within the other parts of the trace generator (sections 6.3.3 and 6.3.4).

Both types of modification are actually minimal, since the MCGREP-2 API (section 6.2.4) is the same for all ISAs. The API includes functions for generating microprograms and decoding instruction words. The operation of these functions is dependent on the ISA and the MCGREP-2 configuration, but the way in which they are used is the same for every ISA.

However, this independence does not account for some of the differences between the Microblaze and ORBIS32 ISAs. One difference is that Microblaze branch operations evaluate a comparison, while ORBIS32 branch operations expect comparisons to be evaluated by a preceding “set flag” instruction. This requires changes to support an alternate type of branch within both the front end and the output stage. These are enabled by a preprocessor macro when the Microblaze ISA is in use. Another difference is that Microblaze branch operations aren’t necessarily followed by a delay slot. This is handled by modifications in the front end, again enabled by a preprocessor macro.

As for ORBIS32, some Microblaze instructions cannot be traced. These include computed jumps, division, and instructions that access special purpose registers. But they also include operations that use the carry flag. This may seem a strange choice, given that many Microblaze instructions do use the carry flag (Table D.1). However, it is non-trivial to accommodate carry flag support within MCGREP-2 traces since additional data paths are needed to transport and store carry flags. Because carry flags are rarely used in compiled Microblaze code, and because the trace generator can already handle unsupported operations correctly using an exit, it was decided to avoid implementing carry flags within traces. (Carry operations work correctly outside of traces.)

D.5 Repeating Experiments and Tests

The components described in sections D.1 through D.4 allow the experiments from section 6.4 to be repeated for a Microblaze ISA version of MCGREP-2. However, not all of the tests can be repeated as there is currently no Microblaze equivalent of the OpenRISC simulator `orlksim`. Therefore, the test that verifies MCGREP-2 against a reference implementation has to be replaced with another

type of test. The replacement for `orlksim` is two test programs, each generated automatically using pseudo-random numbers. The programs exercise the following functions of the Microblaze ISA:

- Instructions: `idiv`, `idivu`, `cmp`, `cmplu`, `sra`, `src`, `srl`, `sxt8`, `sxt16`, `add + rsub` (all types), `and`, `andn`, `or`, `xor`.
- C runtime: division and modulo operations within C.

The test cases used by these programs are implemented from the Microblaze ISA manual [286], and all tests pass on a genuine Microblaze CPU. The tests can be executed as a standalone program in hardware, or on the MCGREP-2 simulator, or both using the `hardware_test` program (section 6.4.1). They provide verification for the Microblaze ISA implementation of MCGREP-2, since implementation errors cause the tests to fail.

The experiment in section 6.4.8 can be repeated using the Microblaze ISA to compute the available WCET reduction and verify the correct operation of the trace generator. The results of the repeated experiment are shown in Figure D.1. (Benchmarks requiring hotspots to be marked using MCGREP-1-style machine code fragments have been omitted.) The results indicate that similar WCET reductions are available with the Microblaze ISA. This suggests that it is very likely that the results obtained with ORBIS32 in chapters 7 and 8 could also be obtained with the Microblaze ISA, although this would require changes to be made to the T-graph building program to support the new ISA (section 7.2.1).

D.6 Booting uClinux

One goal of emulating the Microblaze ISA is support for operating systems. uClinux was used to test this functionality. The Petalinux [197] distribution of uClinux for Microblaze CPUs was used. Petalinux includes the uClinux kernel and user applications along with hardware designs for the Xilinx EDK environment.

Petalinux version 0.10 was compiled for the Xilinx ML401 prototyping board, and installed within the Flash memory of a sample board. A binary image of the Flash memory was then obtained using a uClinux program. The MCGREP-2 simulator's internal memory map was extended to replicate the hardware architecture assumed by Petalinux (Table D.3): the Flash memory image was installed at address `0x28000000`. A tiny boot program containing only a branch to address `0x28000000` was used to start the uClinux boot process within the MCGREP-2 simulator. The boot process began correctly, but was found to stall during attempted initialisation of the Ethernet driver, which is not simulated. Subsequent tests found that disabling support for Ethernet as part of a custom kernel configuration would allow the boot process to complete.

By generating a CPU with an OPB connection, the hardware can be used as a plug in replacement for Microblaze within EDK. This allowed Petalinux to boot on ML401 hardware using an MCGREP-2 CPU. Although this CPU is not fully compatible with Microblaze, the Petalinux system appeared to operate correctly. It was possible to log in to Petalinux, execute uClinux versions of the Microblaze ISA tests (section D.5), and transfer data via Ethernet. However, the system was significantly slower than the Microblaze system. The bus latency of the Petalinux system is at least ten clock cycles for accesses to SRAM, and in the absence of any scratchpads or caches, this memory bottleneck puts a hard limit on the speed of the CPU. This highlights the importance of

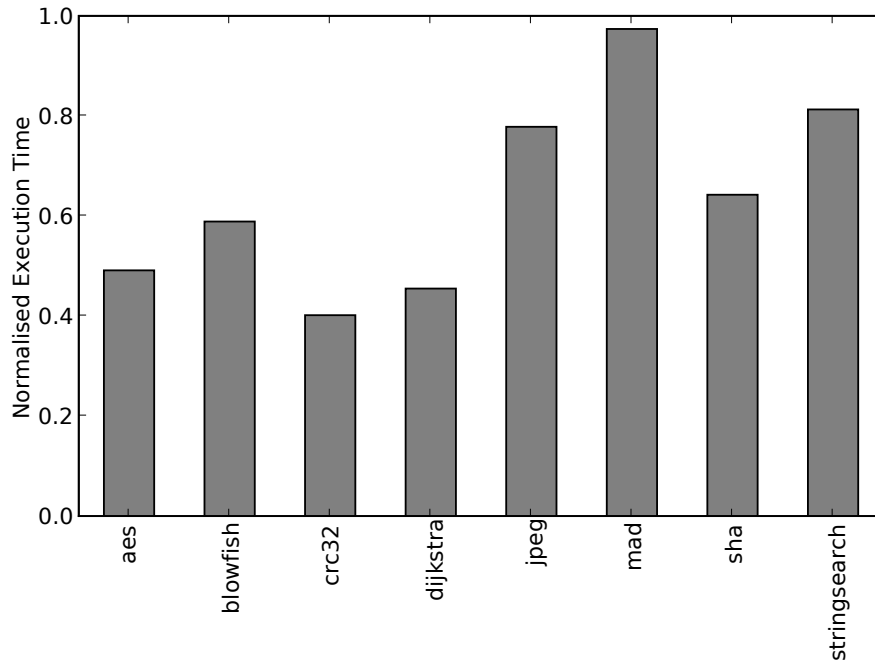


Figure D.1: Normalised benchmark execution times on MCGREP-2 CPUs with traces, using the Microblaze ISA. The MCGREP-2 CPU configuration used here has 3 units and space for 512 microinstructions, and each result is normalised to the execution time of the same benchmark using Microblaze machine code only.

Microblaze ISA Tests

▷ /mcgrip2-src/mcgrip/tests/
ISA test generators and compiled code.



Memory Area	Device	Sim?
0x0000XXXX	Boot RAM	Yes
0x1XXXXXXXX	DDR memory (kernel is loaded here)	Yes
0x200XXXXXX	SRAM (not used by Petalinux)	Yes
0x28XXXXXXXX	Flash (initialised from image)	Yes
0x6000XXXX	Ethernet	No
0x90000XXX	GPIO device	No
0x90001XXX	GPIO device for expansion header	No
0x90002XXX	GPIO device for LCD	No
0x90003XXX	GPIO device for reset	No
0xA000XXXX	Serial driver	Yes
0xA20000XX	Timer 0	Yes
0xA50000XX	USB device	No
0xA60000XX	AC97	No
0xA900XXXX	PS2	No
0xCF000XXX	System ACE	No
0xD0000XXX	OPB/DCR Bridge	No
0xD1000FXX	Interrupt controller	Yes
0xFFFE80XX	Microblaze debug module	No

Table D.3: Memory map for Petalinux version 0.10 for the Xilinx ML401 prototyping board. Some of the devices can be simulated by extensions to the MCGREP-2 simulator.

MCGREP-2 uClinux support

▷ /mcgrep2-src/mcgrep/architectures/microblaze/lowlevelsim.py

Simulator extensions for devices listed in Table D.3.

▷ /mcgrep2-src/scripts/mcgrep_make_opb_vhdl.py

Generates an MCGREP-2 CPU with OPB connection, suitable for use within Xilinx EDK.



solving the memory bottleneck problem before solving the instruction rate bottleneck problem in a practical system.

Appendix E

Implicit Path Enumeration Technique - Example

This appendix gives a worked example for the *implicit path enumeration technique* (IPET), as described in section 2.1.2 and extended in section 7.1. The example is based on the work of Puschner and Schedl published in [207]. The IPET process involves the following steps:

1. For the program being analysed, obtain the T-graph $G = (V, E)$. The method used for this step is not specified. It depends on the implementation of the program to be analysed. A sample T-graph is shown in Figure E.1. In the graph, each edge $x \in E$ has an associated flow variable $f(x)$ and an associated cost $\gamma(x)$. Each flow variable $f(x)$ represent the maximum number of executions of edge $x \in E$ along the worst-case path through the program, this being the path that maximises the execution time.
2. Obtain the execution time cost $\gamma(x)$ for all $x \in E$. Table E.1 lists the $\gamma(x)$ values for the T-graph shown in Figure E.1.
3. Generate conservation of flow constraints to represent the structure of the T-graph. These state that the flow into each vertex is the same as the flow out of it. Table E.2 shows the flow constraints for the T-graph shown in Figure E.1. To avoid the need to handle the root and

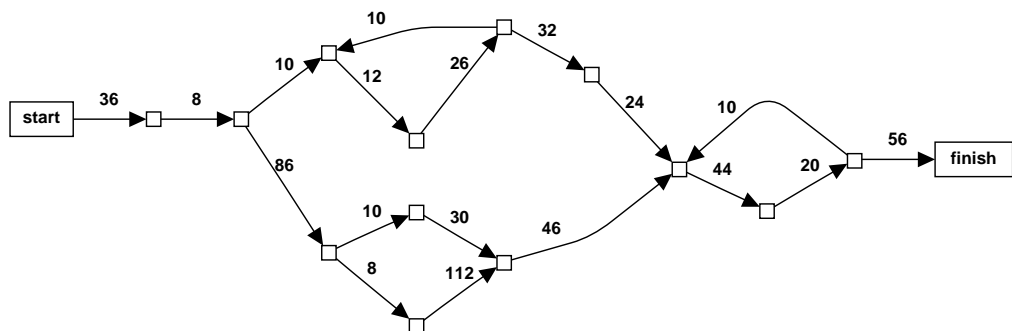


Figure E.1: Sample T-graph, based on an example from [207].

$x \in E$	$\gamma(x)$
x_1	36
x_2	8
x_3	86
x_4	8
x_5	112
x_6	10
x_7	30
x_8	46
x_9	10
x_{10}	12
x_{11}	26
x_{12}	10
x_{13}	32
x_{14}	24
x_{15}	44
x_{16}	20
x_{17}	10
x_{18}	56

Table E.1: Execution time costs for the sample T-graph, Figure E.1.

leaf of the T-graph as special cases, a back edge can be added to link the two with flow 1 and execution cost zero.

4. Add relative capacity constraints [207] to relate the flow within a loop to the flow entering it. The T-graph of Figure E.1 is an example from [207], which specifies the following relative capacity constraints for the two cycles present in the T-graph:

$$f(x_{10}) \leq 8f(x_9) \quad (\text{E.1})$$

$$f(x_{15}) \leq 10f(x_8) + 10f(x_{14}) \quad (\text{E.2})$$

5. Add developer-specified constraints to specify feasible paths through the program. No such constraints are specified in the example from [207]: if they were specified, they would take the same form as the relative capacity constraints.
6. Add non-negativity constraints:

$$\forall x \in E, f(x) \geq 0 \quad (\text{E.3})$$

7. Add the objective function:

$$Z_G = \sum_{x \in E} \gamma(x)f(x) \quad (\text{E.4})$$

8. Translate all constraints and $\gamma(x)$ values into the format used by an integer linear program solver, and instruct the linear program solver to maximise the objective function. Figure E.2 shows the input code for the GLPK linear program solver (section 7.2.3).
9. The linear program solver produces all values of $f(x)$ and Z_G , the WCET for the given T-graph and constraint set. These values are shown in Table E.3.


```

/* variable declarations and non-negativity constraints */
var fx1, integer, >= 0 ;          var fx2, integer, >= 0 ;
var fx3, integer, >= 0 ;          var fx4, integer, >= 0 ;
var fx5, integer, >= 0 ;          var fx6, integer, >= 0 ;
var fx7, integer, >= 0 ;          var fx8, integer, >= 0 ;
var fx9, integer, >= 0 ;          var fx10, integer, >= 0 ;
var fx11, integer, >= 0 ;         var fx12, integer, >= 0 ;
var fx13, integer, >= 0 ;         var fx14, integer, >= 0 ;
var fx15, integer, >= 0 ;         var fx16, integer, >= 0 ;
var fx17, integer, >= 0 ;         var fx18, integer, >= 0 ;
var ZG, integer, >= 0 ;
/* conservation of flow */
s.t. v1: fx1 = fx2 + fx9 ;
s.t. v2: fx2 = fx3 ;
s.t. v3: fx3 = fx4 + fx6 ;
s.t. v4: fx4 = fx5 ;
s.t. v5: fx6 = fx7 ;
s.t. v6: fx5 + fx7 = fx8 ;
s.t. v7: fx9 + fx12 = fx10 ;
s.t. v8: fx10 = fx11 ;
s.t. v9: fx11 = fx12 + fx13 ;
s.t. v10: fx13 = fx14 ;
s.t. v11: fx8 + fx14 + fx17 = fx15 ;
s.t. v12: fx15 = fx16 ;
s.t. v13: fx16 = fx17 + fx18 ;
/* conservation of flow at root and leaf */
s.t. root: fx1 = 1 ;
s.t. leaf: fx18 = 1 ;
/* relative capacity constraints */
s.t. r1: fx10 <= 8 * fx9 ;
s.t. r2: fx15 <= 10 * fx8 + 10 * fx14 ;
/* the objective function */
s.t. z: ZG = 36 * fx1 + 8 * fx2 + 86 * fx3 +
          8 * fx4 + 112 * fx5 + 10 * fx6 + 30 * fx7 +
          46 * fx8 + 10 * fx9 + 12 * fx10 + 26 * fx11 +
          10 * fx12 + 32 * fx13 + 24 * fx14 +
          44 * fx15 + 20 * fx16 + 10 * fx17 + 56 * fx18 ;
/* commands to solver */
maximize obj: ZG ;
solve ;

```

Figure E.2: GLPK solver commands for the IPET example. The results of running this program in GLPK are shown in Table E.3.

Vertex	Constraint
1	$f(x_1) = f(x_2) + f(x_9)$
2	$f(x_2) = f(x_3)$
3	$f(x_3) = f(x_4) + f(x_6)$
4	$f(x_4) = f(x_5)$
5	$f(x_6) = f(x_7)$
6	$f(x_5) + f(x_7) = f(x_8)$
7	$f(x_9) + f(x_{12}) = f(x_{10})$
8	$f(x_{10}) = f(x_{11})$
9	$f(x_{11}) = f(x_{12}) + f(x_{13})$
10	$f(x_{13}) = f(x_{14})$
11	$f(x_8) + f(x_{14}) + f(x_{17}) = f(x_{15})$
12	$f(x_{15}) = f(x_{16})$
13	$f(x_{16}) = f(x_{17}) + f(x_{18})$

Table E.2: Conservation of flow constraints for the sample T-graph, Figure E.1. These describe the structure of the T-graph to the integer linear program solver.

$x \in E$	$f(x)$
x_1	1
x_2	0
x_3	0
x_4	0
x_5	0
x_6	0
x_7	0
x_8	0
x_9	1
x_{10}	8
x_{11}	8
x_{12}	7
x_{13}	1
x_{14}	1
x_{15}	10
x_{16}	10
x_{17}	9
x_{18}	1
Z_G	1262

Table E.3: Worst-case flow values and WCET Z_G for the example.

Appendix F

WCET Reduction Algorithm - Example

Consider the C program in Figure 7.7. Compiling this program for a RISC-like architecture results in the basic block graph shown in Figure 7.12. The time cost of each basic block $\gamma(x)$ is measured by execution: these are invariants because dynamically adaptive CPU features are replaced by an RFU and microprogram store. A constraint set $\{\text{BB6} \leq 100, \text{BB5} \leq 9900, \text{BB4} \leq 5000\}$ is added to represent worst-case behaviour.

The algorithm shown in Figure 7.5 is executed. The first algorithm iteration sets $h = 1$. Using IPET, the original WCET is calculated as $Z_{G_0} = 1323607$ clock cycles.

The next stage of the algorithm adds candidate traces to the program based on the score heuristic (Figure 7.3). The scores for $\Theta = \emptyset$ are shown in Table F.1. The algorithm evaluates W traces of length L in descending order of score, beginning with BB3. For each trace, this evaluation involves *Find_WC_Path* (Figure 7.4), followed by *Generate_Trace*, followed by WCET calculation. For the purposes of this example, the parameters are $W = H = 3$ and $L = 4$.

Given these settings, the WC path from BB3 of length $l = L = 4$ is [BB3, BB4, BB5, BB3, BB4, BB5]. The trace that is generated for that path is shown in Figure F.1. The generator also produces execution time costs for each possible path through the trace (Table F.2), and finds the total space cost of the trace (83 microprogram store lines). The WCET is then calculated: it is reduced to 732707, so the trace benefit is 590900.

The algorithm then proceeds as described earlier. It also evaluates traces beginning at BB5 (benefit 990100) and BB4 (benefit 184800). Of these, BB5 gives the highest benefit, so it is chosen as a candidate and reevaluated for each length $l \in [1, L]$ (Table F.3). The trace beginning at BB5 with length 4 is added to Θ , as it provides the best benefit with the lowest cost. The algorithm recomputes each score in the presence of this trace, then continues with $h = 2$.

Other traces are evaluated. In the second iteration, start points BB1 (benefit 6100), BB2 (benefit 2900) and BB3 (benefit 4900) are tested. The best is BB1, with length 2. In the third iteration, start point BB6 (benefit 300) is tested. The best is BB6, with length 1. Because this example is very small, there is space for all three traces in the microprogram store. For larger programs, solving the extended knapsack problem would eliminate traces with marginal benefits. In this case, all three can be included. The resulting WCET is computed as 322807. Further reductions are possible by increasing L : for example, setting $L = 20$ widens the search and results in a WCET of 190639. However, adjusting H and W makes very little difference to this small program because practically all WC execution occurs in the [BB3, BB4, BB5] inner loop. There are no significant WC paths outside that loop.

Basic block $b(x)$	Exec. cost $\gamma(x)$	WC flow $f(x)$ ($\Theta = \emptyset$)	Score (Figure 7.3)
BB3	81	9900	1.54e+06
BB5	33	9900	1.12e+06
BB4	37	5000	7.50e+05

Table F.1: Possible trace starting points for Figure 7.7, sorted by Score. Costs are given in clock cycles, obtained using the MCGREP-2 simulator (section 6.2.5).

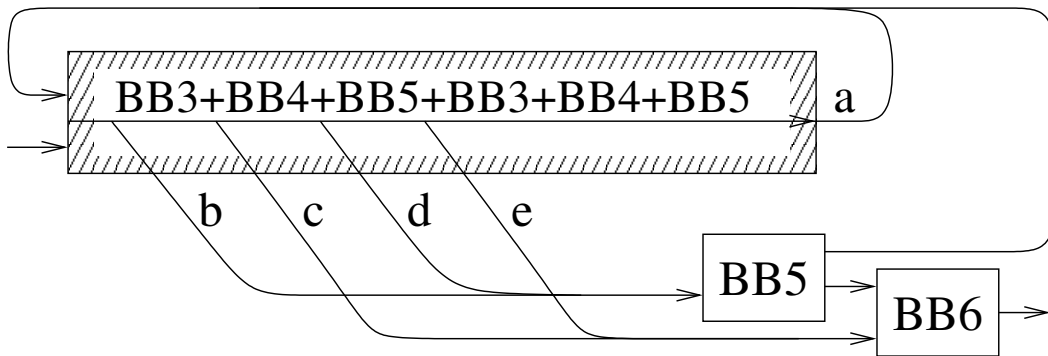


Figure F.1: Trace with length 4 and start point BB3 for the example program (Figure 7.7).

Exit Point	Exec. Cost
a	$\gamma(P_{BB3,a}) = 37$
b	$\gamma(P_{BB3,b}) = 40$
c	$\gamma(P_{BB3,c}) = 48$
d	$\gamma(P_{BB3,d}) = 54$
e	$\gamma(P_{BB3,e}) = 61$

Table F.2: Execution costs (clock cycles) for each path through Figure F.1.

Length l	WCET Reduction
0	29700
1	830900
2	661800
3	990100

Table F.3: WCET reductions obtained with different trace lengths beginning at BB5.

Appendix G

Partitioning Support for the WCET Reduction Algorithm

The TARGET architecture can scale to support programs of any size. This is possible by applying overlaying techniques [191] to partition programs into regions (section 4.3.4), which is supported in implementations of TARGET: section 6.4.10 demonstrated that it is possible to reload the microprogram store during execution, showing that all of the necessary support is available in the MCGREP-2 hardware and software. However, partitioning should also be accommodated by the allocation algorithm. It is possible, but not optimal, to consider each of the regions as a separate allocation problem. If this is done, then any scratchpad or microprogram store allocation algorithm can be used, but the loading time incurred by entering each region will not be properly considered during analysis.

Therefore, partitioning support is implemented as an extension to the algorithms described in chapter 7. In this appendix, section G.1 describes extensions to the algorithms described in sections 7.1.9 to 7.1.12 to support partitioning. Next, a simple partitioning strategy is described in section G.2. Finally, an experiment is used to compare the extended algorithm with scratchpad allocation algorithms in section G.3.

G.1 Adding Partitioning Support For Microprograms

In this section, no assumptions are made about the mechanism that is used to determine region boundaries. The set of all regions is R : each $r \in R$ is a set of basic blocks. R is assumed to be known before allocation begins. The members of R are assumed to be non-overlapping and also complete, i.e.:

$$\forall r_1, r_2 \in R \forall x \in E. x \in r_1 \wedge x \in r_2 \Rightarrow r_1 = r_2 \quad (\text{G.1})$$

$$\forall x \in E \exists r \in R. x \in r \quad (\text{G.2})$$

(This makes R a *set partition* [274] of E , the set of all basic blocks in the program G .)

Two minor changes are needed to extend the microprogram store allocation algorithm with support for R . These are (1) extending *Find_Candidates* so that loading times are modelled at each place where execution crosses from one region to another, and (2) modifying the algorithm described in section 7.1.12 so that each allocation is local to one region.

The first change is accommodated simply by adding a new type of object that can be a member of Θ . This is L_e , the loading cost incurred by a WCET reduction decision at edge $e \in E$. e may be

the starting point of a trace, or a single basic block being allocated to an instruction scratchpad. The new function *Model_Load_Costs* is added to generate L_e for a trace or basic block. The effect of L_e is to increase the cost of all of the entrances to the region containing e by the number of clock cycles required to transfer the trace (or basic block) at e . The effects of this change on *Find_Candidates* are shown in Figure G.1.

The second change is accommodated by repeating the existing algorithm (section 7.1.12) for each region $r \in R$. This processes every region separately, but because loading times are modelled by *Find_Candidates* and incorporated into its WCET reduction estimates, the algorithm balances WCET reductions against additional loading costs.

These changes have two consequences. Firstly, the total size of the traces in Θ may be higher than the total size of the microprogram store or scratchpad. (Reloading makes this possible: only the size of the subset of Θ applying to each region is restricted.) Secondly, the microprogram store and scratchpad may not be completely full, since a trace or basic block will only be selected by the allocation process if the WCET reduction is greater than the worst-case loading time.

The resulting region-aware microprogram store allocation algorithm can also be used as a region-aware instruction scratchpad allocation algorithm. The similarities between Suhendra's algorithm and the microprogram store allocation algorithm have already been explored (section 7.4.1): the same comments apply to the improved algorithm discussed in this section. With the appropriate restrictions (e.g. $L = 1$) the new algorithm is a region-aware version of Suhendra's algorithm.

G.2 An Automatic Partitioning Strategy

The previous section assumed that a partitioning process has already been applied to a program before the microprogram store allocation process begins. In this section, one possible automatic partitioning strategy is described. For a program G , this strategy obtains a region set R satisfying equations G.1 and G.2. Partitioning could be regarded as the first step of scratchpad allocation, with selection being the second step in which code is chosen for migration into the scratchpad.

The basis for the algorithm described here is the Puaut and Pais partitioning algorithm from [202, 203]. An alternative would be manual partitioning using copy points [240], and more complex automatic partitioning approaches are also possible [201], but the use of the Puaut and Pais algorithm matches the choices used within previous work and the requirement for automatic optimisation (section 3.2).

Unfortunately, set partition selection is a complex problem. Optimal partitioning problems are generally NP-hard [209], and program partitioning is no exception [191]. Heuristics may be used to place regions around areas with high worst-case flow, but it is difficult to determine the optimal size for a region. If regions are too large, then more code will run from external memory. This will increase the WCET as external memory is slower than scratchpad memory. However, if regions are too small, then code execution will cross region boundaries more often. Since each region boundary crossing triggers a load, this will also increase the WCET.

Puaut and Pais make use of loop preheaders to mark region boundaries [202]. (These are basic blocks that immediately precede the entrance of a loop.) The intuition is that the code in a loop X is a better candidate for scratchpad allocation than the code immediately outside of X , since the basic blocks in X are executed n more times than the basic blocks immediately outside. This is not worthwhile for all loops:

```

procedure Find_Candidates( $G, R$ ):
  invariant  $G$  = program as T-graph [207]
  constant  $H$  = max. number of WC paths to find
  constant  $L$  = longest trace to consider
  constant  $W$  = number of starting points to consider per step
   $\Theta = \emptyset$ 
   $D = []$ 
  for  $i$  from 1 to  $H$ :
     $Z_0 = \text{Calculate\_WCET}(G_{\Theta})$  // see sections 7.1.2 through 7.1.8
     $(V, E) = G$  // members of  $E$  represent basic blocks
    Compute_Scores( $G$ ) // see Figure 7.3
    sort  $E$  in order of  $e$ .score
    for  $j$  from 1 to  $W$ :
      repeat:
         $e = \text{pop}(E)$  // get "highest scoring" block in  $E$ 
      until  $e$  is a valid start point for a trace
       $P_e = \text{Find\_WC\_Path}(G, e, 0, L)$  // see Figure 7.4
       $T_e = \text{Generate\_Trace}(P_e)$  // see section 6.3
    (1)  $L_e = \text{Model\_Load\_Costs}(T_e)$ 
         $Z_e = \text{Calculate\_WCET}(G_{\Theta \cup \{T_e\} \cup L_e})$ 
         $B_e = Z_0 - Z_e$  // find benefit of this trace
      end for
      find  $e$  such that  $B_e$  is maximised
      for  $k$  from 1 to  $L$ :
         $P_{e,k} = \text{Find\_WC\_Path}(G, e, 0, k)$ 
         $T_{e,k} = \text{Generate\_Trace}(P_{e,k})$ 
    (2)  $L_{e,k} = \text{Model\_Load\_Costs}(T_{e,k})$ 
         $Z_{e,k} = \text{Calculate\_WCET}(G_{\Theta \cup \{T_{e,k}\} \cup L_{e,k}})$ 
         $C_{e,k} = \text{space cost of trace } T_{e,k}$ 
         $B_{e,k} = Z_0 - Z_{e,k}$  // benefit of this trace
        if  $B_{e,k} > B_e$  then:
           $B_e = B_{e,k}$ 
           $T_e = T_{e,k}$ 
        end if
      end for
       $\Theta = \Theta \cup \{T_e\} \cup L_{e,k}$ 
       $D = D + [e]$ 
    end for
  end procedure

```

Figure G.1: The *Find_Candidates* procedure finds and evaluates candidate traces. This version has been extended to support multiple regions and model loading costs. Changes have been made on the lines marked with (1) and (2).

1. The maximum loop count n must be large enough that the reduction in WCET is greater than the worst-case loading time.
2. If X is itself within a loop Y , then a greater WCET reduction may be obtained by placing the load outside of Y , although this depends on the sizes and execution frequencies of both loops.

This region division strategy tends to create regions that are much larger than the loops themselves unless region boundaries are also introduced immediately after each loop exit, which can lead to all code within nested loops being placed in a single region. In fact, without additional region boundaries at loop exits, the strategy simply divides a program into a sequence of outermost loops.

Therefore, in this implementation of the partitioning algorithm, loop exits and entrances (“loop boundaries”) *both* act as region boundaries. The criteria for selecting a loop boundary as a region boundary is the number of times that it is crossed in the initial WC path: i.e. if x is a loop boundary, then x is also a region boundary if and only if $f(x) \leq F$, where F is a positive integer threshold for the worst-case flow.

The strengths of this approach are that it tends to generate region boundaries at rarely executed points in each program (minimising load time), and it is able to create large numbers of regions within a sufficiently large program. It also handles nested loops well: the inner loop will always have a greater worst-case flow than the outer loop, so unless F is large, only the outer loop will be selected as a region boundary. In this regard, F can be seen as a form of resolution setting, with larger F values leading to more regions (fine-grained partitioning) and smaller F values leading to fewer regions, with $F = 0$ leading to exactly one region. F can be adjusted to minimise the final WCET.

One weakness of the approach is that it does not really consider the size of each region, so it is quite possible to end up with a poor-quality partitioning where regions are too small and loading time is significant, or where regions are too big and code on the WC path executes from external memory. Another weakness is that it does not allow commonly-used functions to be present in more than one region. If two loops X_1 and X_2 call the same function X_F , then both loops are placed in the same region. (This problem could be resolved by using virtual inlining during WCET analysis [251].)

All of these problems would need to be solved in order to partition a general program, but partitioning for scratchpad allocation can be effective for small programs [202, 203], so this simple partitioning strategy is sufficient for evaluation purposes. The development of a better partitioning algorithm is outside the scope of this work: more sophisticated WCET-aware partitioning schemes are currently being developed [201], and these could replace the simple scheme described here.

G.3 Comparison with Partitioned Scratchpad Configurations

The extensions to the microprogram store allocation algorithm combined with the Puaut and Pais-based partitioning algorithm (section G.2) make it possible to directly compare the approach published in [202] with the techniques described in chapter 7.

For the following experiment, each benchmark program is tested in two environments: (1) an environment with an instruction scratchpad only, and (2) a hybrid environment with a microprogram store and an instruction scratchpad. The experiment varies the size of each memory. It is assumed that the same number of bits are available in both the scratchpad and the microprogram store.

Because it is not known which partitioning arrangement will be optimal in each case, the experiment tries every possible arrangement that could be generated by the algorithm described in section G.2.

1. **Experiment Goal:** To evaluate an instruction scratchpad approach against a hybrid approach to WCET reduction, using partitioning to make better use of available memory, on different memory sizes. The variables are:

- The allocation algorithm. The region-aware version of *Find_Candidates* (Figure G.1) is used. For environment 1, it is configured to allocate instruction scratchpad space only. For environment 2, it is configured to allocate instruction scratchpad space first, then allocate microprogram store space.
- The scratchpad size. The scratchpad size is varied from 512 bits to 100k bits in order to represent a number of possible configurations. (When a microprogram store is used, it is assumed to be the same size as the instruction scratchpad, as in section 8.3.)
- The benchmark program. The benchmark is taken from a subset of Table 7.2.

The constants are the CPU platform (chapter 6), the memory subsystem and the loading time model. A two-unit MCGREP-2 CPU is assumed.

2. **Software Setup:** For this experiment, no simulation is necessary, since all results can be computed using an IPET-based WCET analyser (section 7.2.2). However, the results can be validated by simulation by inserting breakpoints at each region boundary. When each breakpoint is reached, the simulator increases the total execution time by the number of clock cycles required for the load.

Scratchpad and microprogram store loading is modelled by assuming that the CPU pauses execution when a reload point is reached, then resumes once loading is complete. Load times are obtained using the following equation (from [203]):

$$t_s = t_i + wt_l \quad (\text{G.3})$$

t_s is the total load cost in clock cycles. t_i is the setup cost for a bus transaction, and t_l is the per-word transfer cost. w is the number of 32 bit words to be transferred, i.e. the number of instructions for instruction scratchpad loads, or the number of bits in a microinstruction divided by 32.

Instruction fetches from scratchpad memory take 1 clock cycle. Instruction fetches from external memory take 10 clock cycles (after [203]). However, scratchpad loads from external memory can make use of burst mode, so after an initial setup time of $t_i = 10$ clock cycles, data is transferred at the rate of $t_l = 1$ clock cycle per word.

3. **Method:** For each environment, memory size and benchmark program, the following procedure was carried out:
 - (a) Iterate through all possible values of F and execute the partitioning algorithm (section G.2) to produce a sequence of partitions for each program. For many values of x , the partition with $F = x$ is the same as the partition with $F = x + 1$, so duplicates are discarded from the sequence.

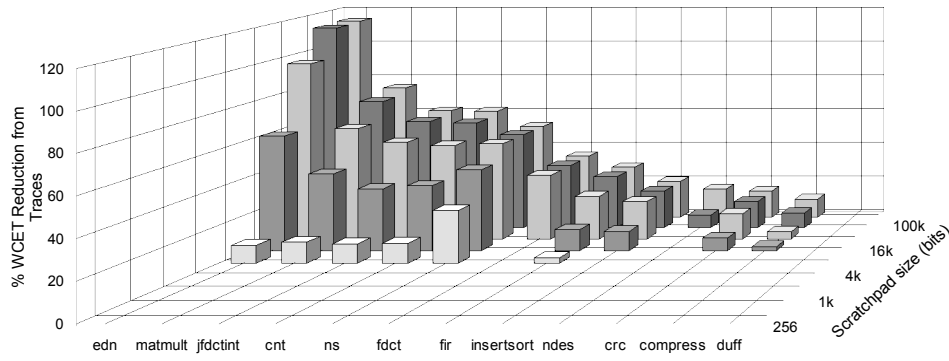


Figure G.2: The percentage WCET reduction δ obtained by using a microprogram store in addition to an instruction scratchpad. The X axis has been sorted in order of the WCET reduction obtained.

Program	256	512	1k	2k	4k	8k	16k	32k
cnt	9	12	4	4	4	1	1	1
compress	12	14	14	12	12	1	1	1
crc	1	3	2	2	1	1	1	1
duff	2	2	2	1	1	1	1	1
edn	22	22	17	17	17	16	1	1
fdct		4	4	4	4	1	1	1
fir	6	6	6	2	2	1	1	1
insertsort	6	6	3	1	1	1	1	1
jfdctint	8	8	8	6	6	6	1	1
matmult	11	11	4	4	4	1	1	1
ndes	16	16	16	16	16	9	1	1
ns	4	2	2	1	1	1	1	1

Table G.1: The number of regions $|R|$ resulting in the optimal WCET for each program in environment 1: instruction scratchpad only. Cells are empty where no WCET reduction could be achieved. Scratchpad sizes are given in bits.

- (b) Apply the allocation algorithm(s) to the partitioned program, either allocating (1) instruction scratchpad space or (2) instruction scratchpad space then microprogram store space, as appropriate for the environment.
- (c) Choose the partition that led to the lowest WCET. Record that WCET, and the number of regions in the partition.

4. **Results:** The results of this experiment are shown in plotted in Figure G.2. The graph shows the difference between environment 1 (instruction scratchpad only) and environment 2 (instruction scratchpad and microprogram store). If the WCET in environment 1 is Z_1 and the WCET in environment 2 is Z_2 , then each data point in the graph is defined by:

$$\delta = 100\left(\frac{Z_1}{Z_2} - 1\right) \quad (\text{G.4})$$

Tables G.1 and G.2 show the optimal number of regions for each environment.

5. **Evaluation:** Figure G.2 shows that the use of custom micrograms can provide a WCET

Program	256	512	1k	2k	4k	8k	16k	32k
cnt	9	12	4	4	4	1	1	1
compress	12	14	14	12	12	1	1	1
crc	1	3	2	2	1	1	1	1
duff	2	2	2	1	1	1	1	1
edn	22	22	17	17	17	16	16	16
fdct		4	4	4	4	1	4	1
fir	6	6	6	2	2	1	1	1
insertsort	6	6	3	1	1	1	1	1
jfdctint	8	8	8	6	6	6	6	6
matmult	11	11	4	4	4	4	4	4
ndes	16	16	16	16	16	9	1	1
ns	4	2	2	1	1	1	1	1

Table G.2: The number of regions $|R|$ resulting in the optimal WCET for each program in environment 2: microprogram store and instruction scratchpad.

reduction when programs are partitioned. Partitioning does not allow the technique to work for very small memories ($< 4\text{kbits}$): when the memory is too small for even a single trace, there is no improvement. Additionally, when loading time is taken into account, some programs (e.g. duff) will not benefit from microprogram store allocation due to the overhead of loading custom microprogram data (these lack sufficient *temporal locality*, section 7.2.7.6). Despite this, experiments indicate that the majority of benchmarks do benefit. As the microprogram store size is increased, the degree of improvement becomes more significant.

Table G.1 indicates that the instruction scratchpad approach benefits from partitioning when the memory size is small, but as the memory size is increased, the benefit of partitioning disappears. With a large memory, everything should stay resident to avoid the cost of loading. Smaller scratchpads benefit from larger numbers of regions because the additional loading cost is completely offset by better use of scratchpad space.

The optimal partitioning is very different for most programs when custom microprograms are in use. Table G.2 shows the optimal region count in this case. Because traces are being used, it is helpful to partition programs like edn and matmult even when the scratchpad is large enough to contain all of the machine code. Partitioning allows more traces to be used, and allows each trace to span more basic blocks. Thus, the WCET can be reduced further.

- Conclusion:** Microprogram store allocation leads to greater WCET reductions than the reductions possible using instruction scratchpads alone, regardless of the algorithm used to allocate the instruction scratchpad.

Once all parts of the WC path through a program are present in an instruction scratchpad, no further WCET reductions are possible: there is a maximum effective scratchpad size for each program. Microprogram store allocation allows the remaining memory to be used to reduce the WCET further by exploiting ILP.

Comparison with Partitioned Scratchpads



▷ `/wctreduce/emsoft.py`

Algo_Compare: carries out the experiment described above.

Index

- ABI, 192
- abstract interpretation, 8
- ACET, 12
- acyclic scheduler, 136
- AES, 39
- AI, 8
- ALE-X, 57
- algorithmics, 38
- application binary interface, 192
- application specific instruction set processors, 50
- application-specific integrated circuit, 45
- Architectural simulators, 97
- ASIC, 45
- ASIP, 50
- average case execution time, 12

- back edge, 142
- basic block, 6
- basic block timing invariance, 12
- biased, 140
- big O notation, 38
- bitstream, 46
- bottom-up greedy, 167
- BUG, 167
- bundle, 135
- burst memory accesses, 14
- busy wait, 49
- byte steering, 105

- cache, 13
- central processing units, 1
- CFG, 214
- CGRA, 55
- Chimaera, 63
- CISC, 15
- CLB, 46
- co-design, 52
- co-processor, 33
- coarse-grained reconfigurable architecture, 55
- codec, 100
- combinational function, 46
- complex instruction set computing, 15
- configurable logic block, 46
- copula theory, 34
- CPU, 1
- critical path, 28, 64
- cyclic scheduling, 144

- DAISY, 19
- dataflow intermediate language, 60
- DCT, 39
- deadlines, 2
- delayed branches, 25
- digital signal processors, 41
- DIL, 60
- direct-mapped cache, 14
- discrete cosine transform, 39
- domino effect, 29
- DSP, 41
- DyC, 44
- Dynamically Architected Instruction Set from Yorktown, 19

- EDA, 45
- electronic design automation, 45
- embedded systems, 1
- EPIC, 18
- explicit path enumeration, 6
- explicitly parallel instruction computing, 18

- fabricate, 45
- fabrication plant, 45
- field-programmable gate array, 46
- fixed priority preemptive scheduling, 35
- floating point operations per second, 13
- FLOPS, 13
- FPGA, 46
- fully associative, 14

- functional simulators, 97
- Garp, 60
- general-purpose registers, 104
- global compaction, 67
- glue logic, 46
- GPR, 104
- graphical user interface, 112
- GUI, 112
- halting problem, 6
- hard and soft configurations, 59
- hard macrocell, 46
- hard real-time, 2
- hardware description language, 44
- hardware description languages, 44
- hardware virtualisation, 53
- hardware/software partitioning, 52
- Harvard architecture, 13
- HDL, 44
- hooks, 159
- horizontal, 66
- horizontal microcode, 66
- hotspot, 40
- hthreads RTOS, 34
- Hyperblock scheduling, 142
- hypervisor, 19
- IC, 12
- IDG, 25
- II, 144
- ILP, 16
- immediate values, 113
- implicit path enumeration technique, 9
- In order, 17
- infeasible paths, 8
- initiation intervals, 144
- input/output block, 46
- instruction dependence graph, 25
- instruction level parallelism, 16
- instruction register, 83
- instruction scheduling, 135
- instruction set architecture, 12
- instructions per clock cycle, 126
- integer linear program, 10
- integrated circuit, 12
- intellectual property, 46
- intelligent devices, 1
- interference, 35
- interrupt service routine, 123
- IOB, 46
- IP, 46
- IPC, 126
- IPET, 9
- IR, 83
- ISA, 12
- JIT, 19
- joint test action group, 47
- JTAG, 47
- just in time, 19
- KressArray, 57
- lambda operation, 44
- least recently used, 14
- load/store unit, 82
- local compaction, 67
- loop kernel, 144
- LRU, 14
- LSU, 82
- machine instructions, 6
- mask programmed gate array, 46
- MATRIX, 58
- MCGREP, 99
- Mei's CGRA, 61
- metaprogramming, 43
- Microblaze, 51
- microchip, 12
- microcode, 65
- microcode compaction, 67
- microinstructions, 65
- microoperations, 67
- microprocessor, 12
- microprogram, 65
- microprogram compaction, 67
- microprogram counter, 111
- millions of instructions per second, 12
- mixed branch predictor, 26
- MMX, 39
- modulo scheduler, 144

multi issue, 16
multi-way associative, 14
multiplexer, 108
multiprocessing, 18
mux, 108

Nanodata QM-1, 66

ODL, 11
OpenRISC, 39
optimisation, 11
optimisation description language, 11
Out-of-order, 17
out-of-order issue unit, 17

PADDI, 59
partial component clustering, 167
partially reconfigurable, 48
path profile, 190
PC, 81
PCC, 167
performance simulators, 97
periodic tasks, 5
photomask, 45
pipelined reconfiguration, 60
pipelining, 16
Piperench, 60
placement, 45
platform, 44
PLD, 46
PR, 48
pragma, 42
precise exceptions, 143
preemption cost, 36
priority inversion, 28
PRISC, 62
probabilistic WCET, 34
profile, 40
program, 1
program counter, 81
programmable logic devices, 46
pseudo round robin, 27
pWCET, 34

Rapid, 59
rDPU, 56

real-time, 2
real-time system, 2
reconfigurable computing, 48
reconfigurable data path units, 56
reconfigurable functional unit, 62
reconfigurable instruction set processor, 62
register clustering, 167
register update unit, 165
registers, 46
REMARC, 58
ReRisc, 64
resource allocation decisions, 29
restricted instruction set computing, 15
RFU, 62
RISC, 15
RISP, 62
routing, 45
RTR, 48
RTS, 2
run-time reconfiguration, 48
RUU, 165

SB, 46
scan chain, 153
schedulability, 5
scratchpad memory, 14
semi-systolic array, 56
set partition, 313
shift register, 47
side entrances, 140
sign extension, 106
SIMD, 18
simultaneous multi-threaded, 19
single instruction, multiple data, 18
single issue, 16
single-path analyses, 32
single-path paradigm, 30
SMS, 144
SMT, 19
software, 1
software pipelining, 144
sorting, 38
sparse matrix multiplication, 43
speculation, 16
speculatively, 16

- SSE, 18
- stalled, 19
- stalling, 17
- standard cells, 45
- standard template library, 38
- STL, 38
- stream processor, 50
- streaming data, 50
- structured ASIC, 46
- Superblock scheduling, 141
- Superscalar, 17
- swing modulo scheduling, 144
- synthesis, 47

- T-graph, 10
- tags, 13
- tail duplication, 141
- task-level parallelism, 18
- tasks, 5
- TDMA, 87
- Tempo, 44
- temporal locality, 222
- threads, 18
- time complexity, 38
- time division multiple access, 87
- time slicing, 19
- timing anomaly, 29
- timing graph, 10
- TLB, 143
- TLP, 18
- trace cache, 14
- trace scheduling, 137
- translation lookaside buffer, 143
- Tree traversal scheduling, 142
- treeregion, 142

- uPC, 111
- user logic, 44
- user microprogramming, 65

- Verilog, 45
- vertical, 66
- vertical migration, 70
- very long instruction word, 18
- VHDL, 45
- virtual FPGA, 53
- virtual inlining, 22
- virtual memory, 53
- virtual simple architecture, 31
- virtualisation, 48
- VISA, 31
- VLIW, 18

- WC flow, 201
- WC path, 32
- WCET, 5
- WCS, 66
- WCTA, 24
- worst case execution time, 5
- worst case timing abstractions, 24
- worst-case execution path, 32
- worst-case flow, 201
- writable control store, 66

- YHAC, 34
- York Hardware Ada Compiler, 34