A SELF-OPTIMISING SIMULATOR FOR A COARSE-GRAINED RECONFIGURABLE ARRAY

Jack Whitham and Neil Audsley
Real-Time Systems Group
Department of Computer Science
University of York, York, YO10 5DD, UK
jack@cs.york.ac.uk

Keywords: CGRA, Simulation

Abstract

This paper describes the implementation and testing of a high-speed simulator for a reconfigurable processor architecture named MCGREP. The architecture is based on a coarse-grained array of small processors controlled by reconfigurable microcode. A high-speed simulator is needed to allow complex experiments to be carried out on MCGREP, involving large applications and time-consuming computations.

Contributions include descriptions of methods for generating, using and testing a simulator for a coarse-grained reconfigurable architecture (CGRA). Special issues that are handled include a requirement to convert MCGREP microcode into native code at any time during execution, and a need to support architectural extensions for future experiments.

1 Introduction

MCGREP [27] is an array of simple processors, organised in the form of a *coarse-grained reconfigurable architecture* (CGRA) [8, 26]. Both fine- and coarse-grained reconfigurable architectures consist of arrays of processing nodes connected by a network of routing elements.

Coarse-grained architectures differ from fine-grained architectures in that each node is capable of operating upon more than a few bits at a time. The reduction in granularity brings a number of benefits, such as easier programmability [8] and fast local reconfiguration [22]. In the MCGREP model, the CGRA can act as a general-purpose processor that interprets RISC machine instructions. However, it can also be programmed with application-specific configurations that can be used to be used to speed up computational bottlenecks (*hotspots*) in applications.

Application-specific function accelerators have long been recognised as a good way to speed up overall performance [18]. These may be included as fixed hardware devices (ASICs), as processor extensions [6], or as run-time reprogrammable devices [10, 14]. Reconfigurable hardware platforms remove limitations on the number of accelerators that can be supported by allowing available hardware to be virtualised.

Research involving MCGREP architectures requires software tools to build CGRA configurations, and working models implemented in either hardware or software for experiments. Simple initial versions of all of these components have been written, but these initial versions are unsuitable for large-scale experiments involving complex applications.

In particular, the MCGREP simulator is not sufficiently fast to execute a Just-in-Time (JIT) compiler. The JIT compiler is used to move pure software functions onto CGRA configurations, effectively optimising an application for the MCGREP platform. Unfortunately, the initial version of the MCGREP simulator cannot execute the JIT compiler in a reasonable time frame, as millions of instructions are needed for each JIT compilation, and the initial simulator executes only a few thousand instructions per second. Thus, the JIT compiler has to be run in a test harness outside of MCGREP, and online "self" optimisation cannot be tested. This is just one example of a complex application that cannot effectively be executed in the initial simulator.

This paper describes the implementation and testing of a new simulator for MCGREP architectures, which has far better performance but retains implementation detail and capabilities for extension and testing. The general contribution is a description of some simple methods for generating, using, and testing a high-speed CGRA simulator that is compiled to native code. The contribution to the MCGREP project is a fast system for automatic testing and future experimentation.

Section 2 describes related work and sources used. Section 3 describes the MCGREP architecture, leading to the description of the initial simulator in section 4. Requirements for the new simulator are described in section 5, followed by implementation details. Section 6 describes the tests used to show correctness, and section 7 evaluates the approaches used against the implementation requirements. Section 8 concludes.

2 Previous Work

Application-specific devices are commonly used in both embedded systems and home computers. Historically, embedded

systems have used CPUs that are optimised for particular task types, such as DSPs, and today some use CPUs that can be adapted to applications, such as Xtensa processors [6]. Home computers also include some application-specific devices such as graphics accelerators.

All of these devices are fixed. If the processors currently given application-specific tasks could be reconfigured, they could be reused by other applications. Hardware devices could be virtualised and instantiated from memory or disk when required. In this way, an effectively unlimited set of configurations could be supported.

The program for a hardware device is called a *configuration*. If this can be changed at all, the device is *reconfigurable*. Some devices may be *run-time reconfigurable*, allowing them to be programmed with new functions during execution.

In section 2.1, reconfigurable hardware approaches are surveyed in general. This is followed by an examination of simulation techniques and existing simulators in sections 2.2 to 2.5.

2.1 Reconfigurable Hardware

Reconfigurable hardware devices are usually built upon an array of configurable processing elements, linked by a communication network.

The most well-known devices are FPGAs, which are intended to be as generic as possible, supporting many different types of logic circuit. These are fine-grained reconfigurable architectures, because most logic elements are as primitive as possible, providing simple combinatorial functions and single-bit registers. FPGA configurations are generated by a complex resource-intensive process that includes heuristic search for placement and routing of elements [3].

Some FPGAs are run-time reconfigurable, and this has been used to create virtual hardware devices. The Molen [10] project exploits run-time reconfiguration, but the configurations must be generated *a priori* using workstation tools. In contrast, the Warp [14] project includes online generation of configurations using a JIT compiler that targets an FPGA. However, this process is still reported to be resource intensive, despite the use of simplified tools [15].

Coarse grained reconfigurable arrays (CGRAs) are similar to FPGAs but the processing elements implement high-level functions directly: often taking the form of *arithmetic-logic units* (ALUs) [8, 26]. This simplifies configuration building, which may avoid heuristic search entirely by use of greedy algorithms (PipeRench [22]) while others may use tree matching algorithms (Garp [9]) or scheduling algorithms (Mei [16]). It also simplifies run-time reconfiguration, which may be even be done on every clock cycle, as in the PipeRench architecture, and in the experimental MCGREP architecture [27].

2.2 Architectural Simulation

Simulation is the process of modelling the behaviour of one system using another. Computers are often used to carry out

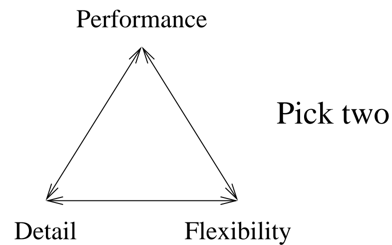

Figure 1: The tradeoffs in simulator design. Performance is speed of simulation. Detail minimises risk from differences between the microarchitecture and the simulator. Flexibility allows experiments with related architectures.

simulations of both natural phenomena and man-made processes to test the eventual results of particular decisions. Simulation allows experimentation with a new design while only parts of the design are complete.

For example, aerospace engineers will experiment with small scale models and computer-generated models of aircraft before building a full-size prototype: this allows early design work to be evaluated before expensive construction work. Similarly, in this application, simulation allows hardware design prototypes to be tested without the expense of building any hardware devices. Simulators also facilitate experiments and automatic testing, because the simulated device can be manipulated by computer programs.

Architectural simulators are programs that execute on a *host* system and provide a model of a CPU. They are often classed as either *functional* or *performance* simulators [23]. Functional simulators are concerned with replicating the functionality of a CPU (the *instruction set architecture*, 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, which allows accurate timing information to be obtained and provides a more detailed simulation.

The tradeoffs in simulator design are illustrated in Figure 1, from [23]. The three factors cannot be optimised simultaneously.

2.3 Existing Simulators

Existing simulators target many common types of CPU, often also simulating memory and some types of input/output device. The SimpleScalar [4] 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 [4, 23], as shown in Table 1. More detailed simulators produce more accurate results, but are slower, allowing the user to choose an appropriate tradeoff from Figure 1.

Simulators are extensively used for embedded development

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

Table 1: The different types of simulator in the SimpleScalar toolset. These simulators are sorted in order of timing accuracy.

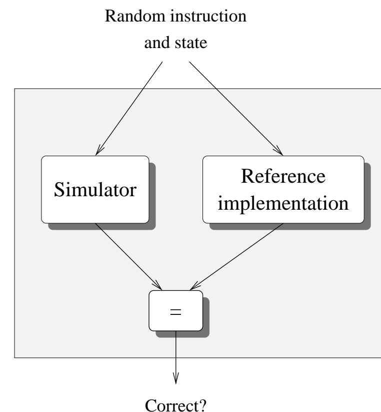

Figure 2: Verification of a simulator by random testing against a reference implementation, as used in the development of SimpleScalar.

work. The OpenRISC development kit [11] includes a functional simulator for the OpenRISC processor, which also simulates some peripherals. The ARMulator is an official performance simulator from ARM Limited [1]. Bochs [12] is a functional x86 simulator for development work.

Simulators are also used for office tasks and entertainment. PowerPC versions of Virtual PC simulate an x86 processor with PC peripherals [2]. The Dosbox [5] simulator provides an DOS-era PC environment for retro gaming. These simulators are functional as high performance is essential for interactive use.

2.4 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 [24] and real hardware. The verification involved random testing, by feeding random instructions and states as input to the simulator and the reference implementation (Figure 2). Performance testing of elements that can't be modified externally, such as caches, can be performed in a similar way - but it may be necessary to use a sequence of instructions (such as a benchmark program) rather than a single one.

2.5 High Performance Simulation

Simulation involves *translating* simulated code or microinstructions into the native code of the host. Usually this is done by *interpretation*: a program examines the simulated code and takes appropriate action for each instruction. For higher performance, *dynamic translation* (JIT compilation) can be used to replace simulated code with native code.

This technique is most famously used by the Java virtual machine [25], as a part of simulating a Java CPU. It is also used by the Bochs x86 simulator and Virtual PC. These simulators normally operate by interpretation, but can run JIT compilation on computational bottlenecks to make the most commonly used code run faster.

3 MCGREP Microcoded CGRA

MCGREP is a CGRA with three levels of configuration. The *high-level configuration* specifies the layout of CGRA nodes and their interconnections. The *mid-level configuration* controls the run-time operation of each node. The *low-level configuration* specifies the operations available at each node. All three of these can be changed at build time, when the processor is created. However, the mid-level configuration can also be reprogrammed during run-time.

The MCGREP architecture is intended to be used to execute the same tasks that are currently given to processors within embedded systems. Experiments are needed to evaluate how closely the architecture's capabilities match up to this goal. Many of these experiments can be carried out very effectively on a simulator, with the added advantage that every level of configuration can be adjusted easily.

3.1 Details

Figure 3 illustrates an array of nine MCGREP nodes, with an interconnection described by the high-level configuration. Figure 4 illustrates a single MCGREP node, with a function set determined by the low-level configuration.

Each MCGREP node contains a processor. Each has a local register file and a time-multiplexed configuration store, as used in the PipeRench [22] processor. This configuration store actually contains a program, specified by the mid-level configuration. This program can be changed at run-time, providing

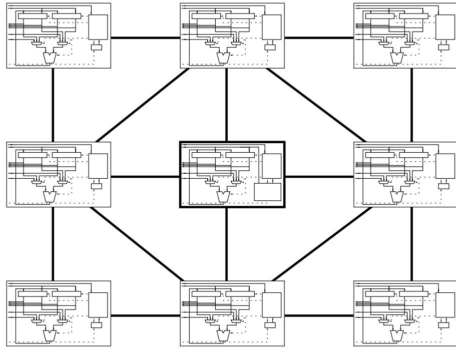

Figure 3: A member of the MCGREP architecture class, with nine nodes.

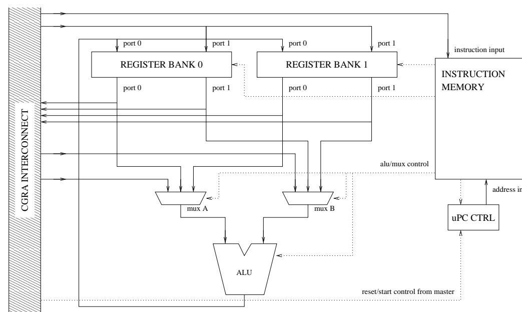

Figure 4: An MCGREP node.

run-time reconfiguration. It is a *microprogram* because it specifies processor operations directly with no layer of abstraction. Instructions within the microprogram consist of *microcode*.

Microprograms have been used to control the internal operations of processors for many years [19] - however, they cannot normally be redefined. Microprograms direct processor operations at a very low level, usually with a direct mapping between microcode bits and control lines.

In MCGREP, each microprogram is a sequence of hardware configurations that carry out some task. New microprograms can be uploaded into any node at any time, allowing the platform to adapt to new applications. Complete reconfiguration of each node takes place every machine cycle, from the microcode configuration.

The physical array in Figure 3 may be dynamically split into *virtual arrays* which cooperate to run a particular task. Virtual array configurations may be produced by any of the CGRA configuration generation techniques described in [8], but the current approach is the use of a modulo scheduler as proposed by Mei [16]. However, the details of this process are outside the scope of this paper.

Virtual arrays may be any size, so they can fit anywhere within the physical array. Physical adjacency is required for cooperating nodes to permit inter-node communication, but this only requires a virtual array to be composed of contiguous blocks.

```
7084: 8c e4 00 00  l.lbz r7,0x0(r4)
7088: 9c a5 ff ff  l.addi r5,r5,0xffffffff
708c: d8 06 38 00  l.sb 0x0(r6),r7
7090: 9c 84 00 01  l.addi r4,r4,0x1
7094: bc 25 ff ff  l.sfnei r5,0xffffffff
7098: 13 ff ff fb  l.bf 7084 <_memcpy+0x1c>
709c: 9c c6 00 01  l.addi r6,r6,0x1
```

Figure 5: A sequence of conventional opcodes (from `memcpy`, which is a hotspot in the `crc32` benchmark program).

3.2 Operation

MCGREP nodes have the ability to run a machine code via an interpreter microprogram. This enables them to execute code that is not specific to MCGREP.

One MCGREP node is active at boot up. This node is referred to as the *local* node for the purposes of this discussion: all others are considered to be *remote*. The local node begins executing machine code from memory in the manner of any conventional processor. There is a three stage pipeline, and an entire RISC instruction set can be supported. This is sufficient to run any operating system or application compiled for the correct instruction set architecture. However, execution speed is bounded by memory latency and no parallelism is possible.

MCGREP currently interprets OpenRISC machine code. Because this interpretation is hardware-assisted, it executes at approximately the same speed as an OpenRISC without an instruction cache [27].

The OpenRISC architecture has reserved some areas of the opcode space for extended instructions. In MCGREP, one of these areas is defined for "jumps into microcode", with the low-order bits of the opcode used to select the microcode address. These opcodes act as a sort of system call, allowing machine code to trigger a microcode function.

To obtain maximum throughput, programs can upload new microprograms into the microprogram store of one or more nodes, and then trigger their execution. This permits application-specific operations to be encoded as single instructions. Such microprograms can span multiple nodes, in order to take advantage of instruction-level parallelism.

This is particularly useful when a hotspot, or computational bottleneck, is reached. This is an area of application code that is commonly used [17]. The efficiency of hotspot execution has a greater effect on an application's execution time than the efficiency of any other code. Figure 5 shows a sample hotspot from `memcpy`.

In MCGREP, hotspot optimisation requires a process to generate microcode from original RISC machine code. Translating a program from one language to a lower level language is normally called compilation: so a *microcode compiler* is needed. The microcode compiler must produce a series of operations that accomplish the same task as the hotspot in less time. Figure 6 shows such a sequence.

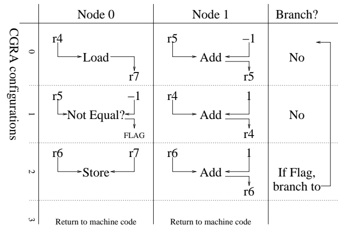

Figure 6: Mapping of Figure 5 onto a sequence of MCGREP configurations.

3.3 Compilation

The details of compilation are outside the scope of this paper. Here, we regard the compiler as a black box which takes machine code as input and emits appropriate microcode for a virtual array of one or more MCGREP nodes. This microcode can then be uploaded to each MCGREP node via writes to a special area of memory.

Early versions of MCGREP used a partially manual process for this compilation, in which operations were scheduled onto a virtual array by the user before microcode was generated. More recently, a JIT compiler has been developed to handle this task automatically, using modulo scheduling techniques as proposed by Mei [16].

Microprograms have many advantages over machine code programs in MCGREP. They may access a much larger register space - currently 512 registers per functional unit, which can be preloaded with constant values that are used by code. They can communicate with other nodes to fetch remote register values. They can execute twice as quickly on each unit if register accesses are optimised. Microbranches are zero-cost. They can also cause microprograms to begin executing on remote nodes by sending a branch instruction to them.

4 Initial Simulator

MCGREP simulators cannot operate at the purely functional level, as some of the simulators described in section 2.3 do. It is necessary to execute microprograms in order to simulate MCGREP, because these can be changed. To do this, it is also necessary to specify the operation of the individual hardware components inside MCGREP, because each bit of the microcode has a meaning only at the lowest level of operation. Thus, the simulator has to be a performance simulator with an accurate model of all parts of MCGREP. In the tradeoff diagram (Figure 1), detail must be selected.

Initial implementations of the MCGREP simulator concentrated on correctness, flexibility and clarity of code before performance. A simple simulator architecture was used, as shown in Figure 7, and the entire implementation was written in the Python language [20]. Python is used to describe the configuration of an MCGREP processor, as it is a highly extensible language that is ideal for this purpose. So it was natural to also use it to build all tools relating to MCGREP, including the simulator. This simplifies interfaces between tools and makes it easy for one tool to reuse components from another.

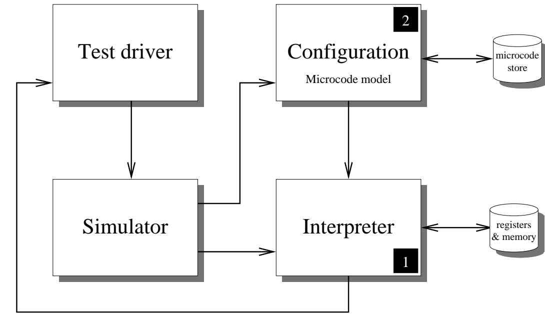

Figure 7: Architecture of initial simulator. Arrows indicate communication between components.

The core of the simulator is the interpreter (labelled “1”). This is called once for every clock tick and every MCGREP node. It fetches appropriate microcode from the configuration (labelled “2”), and then interprets and executes that code. The performance corner of the tradeoff diagram is lost, because each part of the code must be mapped through one or more intermediate codes: this happens very quickly in hardware, where layers of abstraction are optimised away, but requires many operations in software. An example is illustrated in Figure 8.

This problem is exacerbated by the use of the Python language. Unfortunately, Python does not execute directly on the host processor, but via an interpreter of its own (much like Java). Interpreted code is always slower than natively executed code. Conventionally, this problem is avoided by JIT compilation. A JIT compiler for Python does exist [21], but the performance increase from using it in this case is only a 2-3 times improvement.

However, a flexibility advantage comes from the use of Python. Each simulator component is a Python object that can be extended by subclassing. Thus, a basic simulator class type could be extended with a more advanced simulator involving additional features. This was done to create simulators capable of executing advanced tests and for semi-invasive debugging. Such powerful capabilities would need to be retained in some way in more advanced simulators. As in Figure 1, a tradeoff point is reached between performance and flexibility.

5 Implementation

Performance problems in the initial simulator are identified as coming from:-

1. Online interpretation of microcode (*i.e.* microcode is interpreted as it is executed),
2. Online interpretation of the simulator itself.

Resolving these issues is one requirement for the new simulator. However, a more important requirement is the retention of

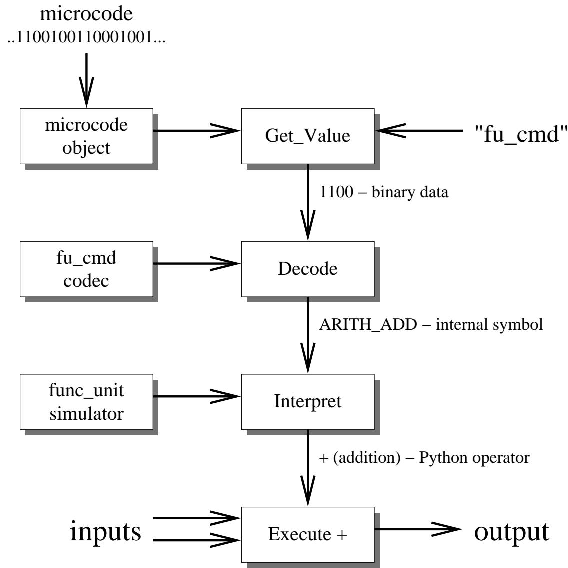

Figure 8: Steps taken to decode an ALU command from the internal representation of microcode to an actual machine operation (addition).

1	Detail	The accuracy of the simulator,
2	Flexibility	The extensibility of the simulator,
3	Performance	The speed of the simulator.

Table 2: Requirements for the new simulator, in priority order. The first requirement has the highest priority.

both flexibility and detail. As Figure 1 and its associated research [23] state, it is not possible to maximise flexibility, detail and performance, so performance can only be improved as long as detail and flexibility are unaffected. Table 2 lists these requirements.

The new simulator architecture is shown in Figure 9. In the new architecture, some components are native code - that is, they are implemented in the native machine code of the host platform. The following sections discuss some of the components of Figure 9.

5.1 Code Generator

The interpreter from Figure 7 is replaced with a code generator (Figure 9, label “1”). This produces C code from microcode descriptions. The code generator is an interpreter, but its interpretations are *offline*: they are fixed code paths as soon as C is generated. The example from Figure 8 is collapsed to a single line of C code to perform an addition. The resulting C program (Figure 9, label “2”) is used for simulation.

This approach avoids both performance problems identified in the introduction. There is no online interpretation. Simulator code is written in the native machine code of the host system. However, flexibility could be lost as any update to microcode must also require an update to the C program.

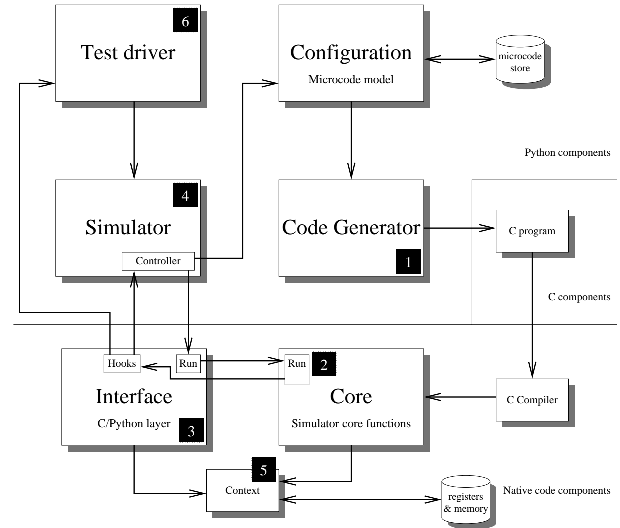

Figure 9: Architecture of fast simulator.

To retain this flexibility, the code generator and the program it produces are both used by a simulation harness (Figure 9, labels “3” and “4”). The simulation harness arranges for the code generator to be reactivated when microcode is updated. The simulated MCGREP can update its own microcode and execute it.

The code generator is an extensible Python object, so adaptations to the architecture can be made by subclassing it and the other code generators it uses. It remains as flexible as the initial simulator.

The code generator could produce machine code directly, as is commonly done by JIT compilers [21]. But this would make the simulator highly specific to a single host architecture, as well as making development and debugging significantly more difficult. Thus, the low-level tasks of code generation are left to a C compiler.

5.2 Core

The simulator core (labelled “2” in Figure 9) presents a number of external interfaces, for communication with the 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. Now that 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. So hooks are provided to formalise the extension process. They are functions that are

Name	Description
`register_file`	General-purpose register values.
`iword`	Instruction word register.
`memory_image`	Simulated RAM.
`upc`	Microprogram counter register - one per node.
`lsu`	Contains state of load-store unit.
`ucode_updates`	List. Stores all writes to microcode for later use by the code generator.
`dirty_table`	Marks dirty microcode (see section 5.4).

Table 4: Principal Context object fields.

```
import SimulatorInterface as SI
try:
    SI.Attach(LIBRARY_NAME)

    context = SI.Make_Context(MEMORY_IMAGE_NAME)

    # No operation (nop) hooks handle special simulator commands
    SI.Set_Hook(context, NOP_Hook, SI.HOOK_NOP)

    running = True
    while ( running ):
        rc = SI.Run_Dynamic(context)
        running = not ( rc in (SI.RC_EXIT, SI.RC_ERROR) )

        if ( running ):
            # Must have hit dirty microcode - update our notion
            # of the microcode store to match simulator.
            Do_Microcode_Update(SI.Get_Microcode_Updates(context))

            # Rebuild library with new microcode, preserving
            # the context (which contains all state data).
            SI.Detach()
            Rebuild_C()         # C program produced
            Rebuild_Library()   # Shared object produced
            SI.Attach(LIBRARY_NAME)
            # Now resume execution with new microcode.

    # Simulation ends
finally:
    SI.Detach()
```

Figure 10: Main loop for simulation. The simulator interface (`SI`) is the component labelled “3” in Figure 9.

called by the simulator core in response to a particular event. These can be used for testing and debugging.

Table 3 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.

An important part of the interface is the context object (labelled “5”). The context object holds all information about the state of an MCGREP processor (as partially listed in Table 4). This object exists independently of the simulator core, so that the core can be updated by the code generator without destroying the processor state.

5.3 Interface

The simulator core is written in C and compiled to machine code, but the rest of the MCGREP tools are written in Python and compiled to Python byte codes. This division is illustrated by the dividing lines in Figure 9.

A glue layer is needed to provide a Python interface to the core.

```
...
EA =0003ffcc PA =0003ffcc r7 =00000030 000064d8 l.sb 0x0(r6),r7
r4 =0000ae1c r4 =0000ae1c              000064dc l.addi r4,r4,0x1
r5 =0000000f                           000064e0 l.sfnei r5,-1
             SR =00008201             000064e4 l.bf -5
r6 =0003ffcd r6 =0003ffcd              000064e8 l.addi r6,r6,0x1
r7 =00000031 EA =0000ae1c PA =0000ae1c 000064d0 l.lbz r7,0x0(r4)
r5 =0000000e r5 =0000000e              000064d4 l.addi r5,r5,-1
EA =0003ffcd PA =0003ffcd r7 =00000031 000064d8 l.sb 0x0(r6),r7
r4 =0000ae1d r4 =0000ae1d              000064dc l.addi r4,r4,0x1
r5 =0000000e                           000064e0 l.sfnei r5,-1
             SR =00008201             000064e4 l.bf -5
...
```

Figure 11: Sample of OpenRISC simulator output (the kernel of the `memcpy` function). The trace allows register writes and program counter updates to be checked.

This is the job of the Interface (Figure 9, label “3”), which translates C calls into a form that can be used directly from Python. It also provides a wrapper to allow hook functions to be written in Python.

This glue layer is written in C, but conforms to the Python specifications for modules written in native code. The Python API is used to translate function calls and variable accesses between Python and low-level C types.

The most essential feature of this glue layer is its ability to load and unload the simulator core program from memory. This is done using the `dlopen` and `dlclose` functions, which permit shared objects (dynamically linked libraries) to be attached to the memory space of a program at runtime. This allows the simulator core to be rebuilt during simulation. These operations do not affect context objects, which can be passed between versions of the simulator core provided that they have been produced for the same revision of MCGREP.

5.4 Simulator

At the top level of simulation, Python code is responsible for recompiling and executing the core. This is done by the controller (Figure 9, label “4”). The controller consists of a main loop, shown in Figure 10, which executes the simulation until an error occurs or the program terminates.

The main loop handles the situation in which *dirty* microcode has been reached: dirty meaning that the representation of the microcode in the C program is out of date. In this event, regeneration and recompilation of the C program is required. These are handled by the `Rebuild_C` and `Rebuild_Library` functions respectively.

6 Correctness Verification

The simulator must operate correctly - that is, it must operate in the same way that a hardware implementation would. This section describes the steps taken to ensure this.

Two distinct types of operation are possible on MCGREP - execution of microcode, and execution of machine code via microcode. Although both involve execution of microcode, these modes are very different. The first type of microprogram is a series of application-specific configurations used to accelerate a task. The second type is an interpreter that fetches values

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 "jump to microcode" (see section 3.2).
HOOK_NOP	Called when an extended nop (no operation) instruction is executed. These are commands to the simulator (*e.g.* exit).

Table 3: Types of hook provided by the simulator.

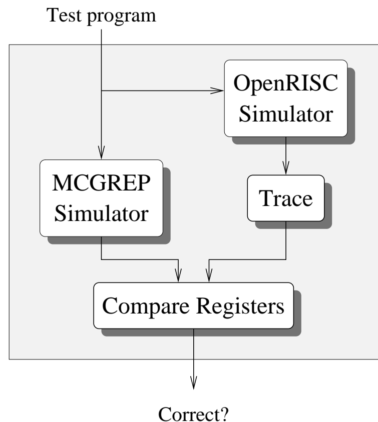

Figure 12: Comparing MCGREP operation to reference OpenRISC simulator trace.

from memory and executes commands based upon them. These are discussed separately.

6.1 Testing the Interpreter

The SimpleScalar simulator was tested by comparison with a reference implementation (section 2.4). In one case, the reference implementation was a real processor, in another, it was another simulator. Similar tests can be performed on the MCGREP simulator.

The interpreter must behave functionally in the same way as an OpenRISC processor [11], so a comparison can be made with the OpenRISC simulator. Exact performance simulation is not required here, but functional correctness is essential: machine code must execute without error.

The OpenRISC simulator can emit a trace file listing all operations executed during the run-time of a test program. These appear in the form illustrated in Figure 11. Operation of the MCGREP simulator can be verified against this, as illustrated in Figure 12.

The comparison is carried out as follows. The interpreter microprogram accesses a dispatch table when it decodes a new instruction. This table gives the appropriate microcode address for handling the current instruction word. At that time, the program counter and the values of registers can be read. A hook function (see section 5.2) is called by the simulator to notify the test driver that a dispatch table access has occurred.

Name	Description
aes	Encryption benchmark.
crc32	Computes CRC-32.
jpeg	Decodes a JPEG image.
sha	Computes SHA-1 hash.
qsort	Sorts a data set.
mad	Decodes MP3 audio.
dijkstra	Shortest-path algorithm.
g721	Audio encoder.

Table 5: List of programs used for testing simulators.

The hook function loads a single line from the trace file and extracts the destination register number and value, if these exist for the current instruction. The program counter is also extracted from the line. The values of the destination register and the program counter are loaded from the MCGREP simulator, and compared against those from the trace.

Various test programs were executed on the simulators for verification purposes (Table 5). These are benchmarks from the MIBench [7] and Mediabench [13] suites. It is not difficult to find a set of programs that will exercise all parts of the MCGREP interpreter program to give full coverage.

6.2 Testing general microprograms

When the initial simulator was written, no reference implementation was available for testing microprogram execution. MCGREP is a new architecture, and it is not compatible with any other type of CGRA. This means that the techniques applied for testing SimpleScalar (described in section 2.4) are not applicable as correct reference behaviour cannot be independently defined.

However, each microprogram does have a well-defined function: it has to do the same job as a particular fragment of machine code (a hotspot). For any particular input condition (register values and memory state), the microprogram must produce the same output condition that would result from running the code (Figure 13).

This is tested by executing two copies of the simulator at the same time. One copy runs a program that has been modified to call microprogrammed versions of its hotspots. The other uses the original machine code. Test programs from Table 5 were used.

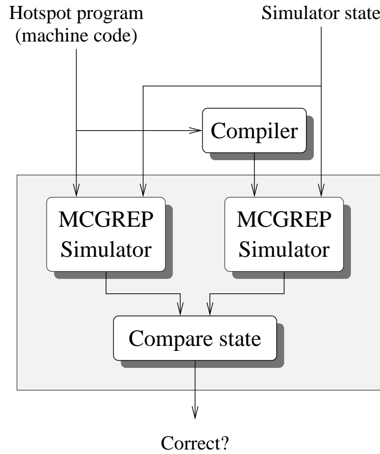

Figure 13: Comparing hotspot microprogram operation with its equivalent machine code in order to validate MCGREP operation. Both simulators are expected to terminate execution in the same state, showing that the microcode implementation of the input hotspot is functionally identical to the machine code implementation.

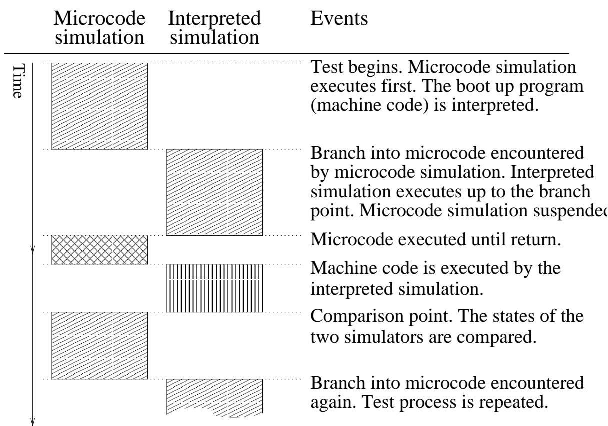

Figure 14: Application of the test illustrated in Figure 13 to an entire program, using two copies of the simulator.

The two simulators are executed according to the pattern shown in Figure 14. This pattern permits state comparison between the memory and registers whenever a microprogram terminates. This comparison assumes that the implementation of the machine code interpreter is correct - but this can be tested separately as described in section 6.1.

The weakness of this technique is that it cannot distinguish between errors in the MCGREP microcode implementation and errors that have been introduced by the compilation process. However, it will always detect errors, which can be resolved by manual debugging.

The new simulator can be tested using this process. However, it can also be compared with the initial simulator, which can act as a reference implementation. This test repeats the process used for testing the interpreter against OpenRISC (section 6.1),

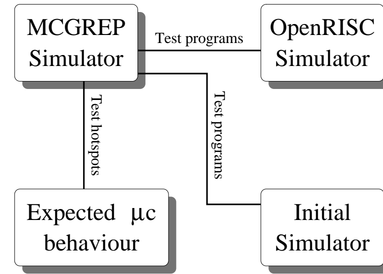

Figure 15: Verification links used for the new MCGREP simulator.

but comparisons are made on every clock cycle instead of every dispatch table access, because execution is now being tested at the microcode level.

6.3 Results

The tests, summarised in Figure 15, successfully validated the MCGREP simulator. Although there is no reference hardware implementation yet, it was possible to test the features of the simulators against other references.

7 Evaluation

Table 2 lists three requirements for the new simulator. Of these, accuracy and flexibility are the most important. Accuracy has already been verified in section 6, as far as is possible without a reference hardware implementation, but what of flexibility and performance?

7.1 Performance Evaluation

The simulator is slowed down by two types of event:

- Calls to hook functions (see section 5.2),
- Execution reaching dirty microcode (see section 5.4).

These events cause execution to exit the simulator core, returning to the host Python environment. Arbitrary Python code may be executed by each hook function. In the case of dirty microcode being reached, a call to the C compiler will also be necessary.

Table 6 gives some data about these speed penalties when the simulator is executing on three reference computers. Performance data for the previous simulator is also listed here.

The hook functions used for Table 6 are empty - they are implemented in Python, but return immediately. This allows the overhead of hook calls to be isolated from the actual operation of a real hook function.

The microcode updates for Table 6 are also minimal. Each is a single write to an extended register, requiring only a few alterations to the microcode store. More complex changes are likely to reduce the speed of compilation further, as there will be more lines of microcode to compile.

Description	Ref 1	Ref 2	Ref 3
No hooks or updates	5,010,000	2,540,000	6,920,000
One microcode update during execution	4,600,000	2,250,000	6,300,000
Two microcode updates during execution	4,210,000	2,040,000	5,780,000
Three microcode updates during execution	3,840,000	1,850,000	5,440,000
One hook call per dispatch	1,100,000	493,000	1,310,000
One hook call per tick	549,000	253,000	647,000
Initial simulator, no hooks or updates	1240	956	2150

Table 6: The effects on simulator performance, measured in simulated clock cycles per host CPU second, running the `sha` benchmark on three reference computers, as dummy hook functions and small microcode updates are introduced. The three computers are (1) Pentium 4 workstation, Linux, 2.8GHz, (2) UltraSPARC-IIIi server, Solaris, 1GHz, and (3) AMD Athlon 64 server, Linux, 2.2GHz.

	Ref 1	Ref 2	Ref 3
Regenerate	0.0667	0.0900	0.0367
Recompile	2.72	6.01	1.54

Table 7: CPU time in seconds required to regenerate and recompile the boot-up microcode configuration of MCGREP on three reference platforms (see Table 6 for details of these).

As these results show, the simulator is normally very fast ($>$ 5 million clock cycles per second on two reference computers). A significant penalty is introduced by any call to hooks or a microcode update. Individual microcode updates are much slower than individual calls to hooks (Table 7), but the sheer frequency of calls to hook functions can overshadow them. Calling a hook function every tick results in an almost ten-fold speed reduction.

Despite this, the common case - uninterrupted execution - is fast. The penalty is regarded as acceptable: especially as the new simulator is approximately three orders of magnitude faster than the initial simulator.

7.2 Flexibility Evaluation

Simulator flexibility falls in to two categories. One is the ability to construct new types of test from the basic simulator. Another is the ability to adjust the simulator configuration: extending the MCGREP array, for example.

Section 6 gives several examples of tests that were performed by extending the simulator. The interpreter test that compared an OpenRISC simulator trace with MCGREP operations used a hook function on the dispatcher. The tests involving machine code and microcode comparisons (illustrated in Figures 13 and 14) also made use of hooks, to carry out the comparisons, set breakpoints, and even to reverse changes made to the machine code to force the two simulators to use different implementations of each hotspot.

That all of these tests were possible without invasive modification of the simulator code is a good indication of the flexibility granted by hook functions. Hook functions even allow some instruction set extensions: the original OpenRISC simulator defines some "extended nop" instructions for host operations like "exit" and "debugging on". These can be supported through a hook function for handling nop instructions - this appears in the main loop, Figure 10.

The simulator configuration is also extensible, but invasive modifications to the code generator are required for some changes. The low level MCGREP configuration specifies the operations available at each node: these are provided by a functional unit generator, written in Python, which produces C code. That generator can be extended by the creation of a replacement Python function.

The high level MCGREP configuration specifies the arrangement of the array and interconnection elements. This can be changed by modifying or extending the high level processor description. Any configuration that is recognised by the code generator can be used. There are currently some practical restrictions regarding the types of connection that are possible, but improvements to the code generator will resolve these.

Because it is possible to change every level of the simulator configuration through extensions to software, and because it is possible to affect simulator execution through hooks, the simulator is regarded as sufficiently flexible for future experiments.

8 Conclusion

The correctness of the simulator has been shown using a variety of tests, including comparisons to other simulators. The hybrid C and Python implementation strategy has allowed flexibility and detail to be retained while performance was significantly improved for the common case. The code generation approach used is also platform independent and relatively straightforward to implement, as a C compiler is used for production of machine code.

This development of a new simulator has also allowed new types of experiment to be carried out using MCGREP. For the first time, the MCGREP simulator is sufficiently fast to be able to execute the MCGREP JIT compiler itself. MCGREP programs have been able to use this to optimise themselves, generating new microcode to accelerate hotspots online. Thus, the goal of this work, enabling further experiments, has already

been reached.

One option for future work is implementation in a hardware modelling language. The accuracy of the simulator could be improved further by a move to an implementation in SystemC, which would permit the interconnections between components to be modelled. However, this would be a return to a fully interpreted simulation, as the type of optimisations created by recompiling the simulator when the microcode changes would not be appropriate in a hardware-level simulation. This is because they involve specialising each device in the data path to the function specified by microcode. A SystemC simulation would give more accurate information about the hardware, at the cost of performance.

References

[1] Anonymous. ARMulator. Application Note 32, ARM DAI 0032F, ARM Limited, 2003.

[2] Apple Inc. Virtual PC for Mac OS X. `http://www.apple.com/macosx/applications/virtualpc/` (accessed 17th Dec 2006).

[3] 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.

[4] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[5] DOSBox Crew. DOSBox, x86 emulator with DOS. `http://dosbox.sf.net/` (accessed 17th Dec 2006).

[6] R. E. Gonzalez. Xtensa — A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, 2000.

[7] 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.

[8] R. Hartenstein. Coarse grain reconfigurable architectures. Embedded Tutorial, ASP-DAC 2001.

[9] J. R. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. In *Proc. FCCM*, page 12. IEEE Computer Society, 1997.

[10] G. Kuzmanov, G. N. Gaydadjiev, and S. Vassiliadis. The Molen Media Processor: Design and Evaluation. In *Proc. WASP*, pages 26–33, September 2005.

[11] D. Lampret. OpenRISC 1200 (accessed 16 Jan 06). `http://www.opencores.org/`.

[12] K. P. Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux Journal*, 1996(29es):7, 1996.

[13] 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.

[14] R. Lysecky, G. Stitt, and F. Vahid. Warp processors. *ACM TODAES*, 11(3):659–681, 2006.

[15] R. Lysecky, F. Vahid, and S. Tan. A Study of the Scalability of On-Chip Routing for Just-in-Time FPGA Compilation. In *Proc. FCCM*, 2005.

[16] 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.

[17] M. C. Merten, A. R. Trick, and R. D. Barnes. An architectural framework for runtime optimization. *IEEE Trans. Comput.*, 50(6):567–589, 2001.

[18] G. D. Micheli, W. Wolf, and R. Ernst. *Readings in Hardware/Software Co-Design*. Morgan Kaufmann Publishers Inc., 2001.

[19] D. A. Patterson and J. L. Hennessy. *Computer organization & design: the hardware/software interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[20] Python Software Foundation. Python Official Website. `http://python.org/` (accessed 29th Nov 2006).

[21] A. Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *Proc. PEPM*, pages 15–26. ACM Press, 2004.

[22] H. Schmit, D. Whelihan, M. Moe, B. Levine, and R. Taylor. PipeRench: A virtualized programmable datapath. In *Proc. CICC*, pages 63–66, 2002.

[23] SimpleScalar LLC. SimpleScalar Hacker's guide. `http://www.simplescalar.com/docs/hack_guide_v2.pdf` (accessed 17th Dec 2006).

[24] SimpleScalar LLC. SimpleScalar Tutorial. `http://www.simplescalar.com/docs/simple_tutorial_v4.pdf` (accessed 17th Dec 2006).

[25] Sun Microsystems. Java Hotspot Server VM: Dynamic Compilation. `http://java.sun.com/products/hotspot/-docs/general/hs2.html` (accessed 29th Nov 2006).

[26] D. Vassiliadis, N. Kavvadias, G. Theodoridis, and S. Nikolaidis. A RISC architecture extended by an efficient tightly coupled reconfigurable unit. In *Proc. ARC*, 2005.

[27] J. Whitham and N. Audsley. MCGREP - A Predictable Architecture for Embedded Real-time Systems. In *Proc. RTSS*, pages 13–24, 2006.