MCGREP - A Predictable Architecture for Embedded Real-time Systems

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

Abstract

Real-time systems design involves many important choices, including that of the processor. The fastest processors achieve performance by utilizing architectural features that make them unpredictable, leading to difficulties proving offline that application process deadlines will be met, in the worst-case. Utilizing slower, more predictable processors, may not provide sufficient instruction throughput to execute all required application processes. This exposes a key trade-off in processor selection for real-time systems: predictability versus instruction throughput.

This paper proposes MCGREP, a novel CPU architecture that combines predictability, high instruction throughput and flexibility. MCGREP is entirely microprogrammed, with multiple execution units. Basic operation involves implementation of a conventional set of CPU instructions in microcode - MCGREP then executes object code suitably compiled. Advanced operation allows the application to dynamically load new microcode, enabling new application specific instructions to increase overall performance.

MCGREP is implemented upon reconfigurable logic (FPGA) - an increasingly important platform for the embedded RTS. Custom microcode configurations for new instructions are generated from C source. MCGREP is shown to have performance comparable to two popular FPGA softcore CPUs (OpenRISC and Microblaze, the latter a commercial product). Flexibility is demonstrated by implementing an existing instruction set (OpenRISC) in microcode, with application-specific instructions to improve overall performance. As a further demonstration, predictable two-level interrupt and synchronization mechanisms are programmed in microcode.

1 Introduction

Real-time systems (RTS) are characterized by the property that *timing behavior* is an essential part of *correctness*. Ultimately, bounded and predictable timing behavior of a RTS is dependent upon the implementation platform (often involving a CPU) to itself to be predictable and bounded from a timing perspective. We observe that RTS are implemented on a wide range of *platforms* (hardware devices), utilizing a wide range of *architectures* (computer system structure as seen at the assembly level). However, all architectural choices for a particular RTS involve tradeoffs. This paper examines some of the fundamental tradeoffs that exist when making an architectural choice for an RTS, and then proposes MCGREP, a CPU architecture that provides high predictability, high performance and high flexibility.

MCGREP is motivated by the need for a CPU that operates predictably: *i.e.* with fixed timing behavior that is unaffected by execution history (*e.g.* cache state). Whilst simple processors already have this property, they are unable to match the performance of complex CPUs as their execution speed is bounded by the speed of memory accesses. More complex CPUs make use of caches and internal tables in order to speed up the average case, but this makes *worst-case execution time* (WCET) analysis more difficult [10].

High performance systems can already be built predictably, but often dispense with the CPU. For example, application specific circuits may be synthesized directly from source algorithms (*e.g.* NISC [19]), the target platform being an *application specific integrated circuit* (ASIC). However, such approaches result in systems without hardware flexibility - the ability to add new features to a system without rebuilding the hardware. For this reason, such approaches are disregarded by this work.

Another approach combines a CPU with custom hardware. An *application specific instruction set processor* (ASIP) [7] tool generates a processor description with hardware support for user functions. The speedup from these can eliminate the need for a cache. ASIP systems are flexible, as software can be changed, but the hardware is fixed.

Other approaches attempt to restrict the use of a fast (unpredictable) processor so that sufficient predictability can be obtained. For example, restricting cache use [3] and monitoring execution times [1] are both applicable. However, in this paper we contend that developing a fast predictable processor is a more appropriate and scalable solution.

MCGREP applies ideas from the field of *run time reconfigurable hardware* to build a CPU that is ideal for real-time systems. In MCGREP key parts of the hardware can be redefined during execution. As in an ASIP, the speedup from this may permit an acceptable operating speed without any need for a cache. However, unlike an ASIP, many different configurations can be stored in program memory.

MCGREP currently targets a *field-programmable gate array* (FPGA) platform, now used as a popular base for small embedded systems due to low cost and customization capabilities. FPGAs are limited in both capacity and speed compared to an ASIC, but are reprogrammable allowing more flexibility [15]. Softcore processors (*e.g.* OpenRISC [12] and MicroBlaze [2]) implemented on FPGAs lack some of the performance of CPUs implemented directly in silicon. However, the overall system performance is comparable, as an FPGA allows many functional units to share a single device (*e.g.* softcore processor, RAM, devices), and allows application specific support to be programmed using logic gates. Additionally, nothing prevents the future implementation of MCGREP in silicon, as MCGREP is not dependent on any FPGA-specific feature.

This paper is structured as follows. The rest of this section summarizes architectural tradeoffs. Section 2 describes the reasoning behind MCGREP, including a review of related work in section 2.3. Section 3 gives an architectural overview and section 4 provides an evaluation against the architectural tradeoffs identified here. Section 5 describes some of the ways in which the flexibility of MCGREP may be used to support an RTS, and section 6 concludes.

1.1 Background

The architecture of a machine includes the *instruction set architecture* (ISA) of the CPU, co-processors, and any other devices connected to the system bus. It is the lowest level seen during programming. The remainder of this section considers the fundamental predictability, performance, flexibility and resource trade-offs when choosing a particular CPU-based architecture for a RTS. The trade-offs are considered across the following five architecture variants:

- **Simple CPU** (example: Motorola 68000)
  A CPU without caches or a complex pipeline.
- **Complex CPU** (example: PowerPC 405)
  A CPU with caches and/or a complex pipeline.
- **ASIP** (example: Tensilica Xtensa [7])
  A CPU with custom extensions. It is assumed that caches and complex pipeline features are turned off.
- **FGRA** (example: Molen [25])
  Fine-grained reconfigurable arrays (FGRAs) extend a conventional processor (assumed to be simple here) with an FPGA-like area which can be programmed with user-specified hardware devices during operation. This permits a program to introduce whatever hardware it requires.
- **CGRA** (example: ReRisc [24])
  Coarse-grained reconfigurable arrays (CGRAs) extend a conventional processor with a network of interconnected functional units that can be reprogrammed to carry out any composite function. Typical functional units carry out simple arithmetic and logic operations. [9] gives an overview of many CGRAs.

1.1.1 Instruction Throughput versus Transistor Count

Instruction throughput is a measure of CPU performance: machine instructions executed per unit time. Throughput is affected by the choice of CPU, the bandwidth of the memory bus, and the clock frequency of the system. For all five classes of architecture, higher throughput costs more transistors. More logic gates and more memory elements are required, as higher throughput is achieved using *pipelines*, *superscalar* execution units, and caches. For an overview of each of these, see [18].

1.1.2 Instruction Throughput versus Predictability

From an RTS perspective, speed versus predictability is an important tradeoff. Architectures featuring complex CPUs may be fast, but are difficult to analyze, and thus difficult to prove safe. This is because of *hidden state*. A CPU component has hidden state if its operation is affected by memory elements that are not directly accessible. Examples of hidden state components include caches and dynamic branch prediction tables. These components trade predictability for average execution speed [26].

Through the hidden state, tasks influence the execution time of other tasks. Even within a single task, interactions between CPU components require complex analysis ([14, 26] provide examples). These interactions makes it impossible to decompose analysis tools into modular, extensible forms [10].

RTS researchers are currently debating the best way to derive tight and safe estimates for WCETs on platforms with hidden state. Two basic approaches exist: CPU modeling to ensure safety [10], or measurement followed by probabilistic modeling [4] to ensure safety within a known probability bound. Hidden state and predictability are not necessarily mutually exclusive, but all approaches for determining WCET become simpler and more accurate when hidden state is reduced.

CGRAs and FGRAs can provide high throughput, but may also introduce unpredictability in two ways. Firstly, the system may be unresponsive during reconfiguration, which defeats analysis approaches that assume that the system is always available for event handling. Secondly, reconfiguration may take an imprecisely known length of time. In particular, run-time reconfiguration of FPGA hardware is neither well documented nor intended to be used in an RTS.

1.1.3 Flexibility versus Resource Efficiency

A flexible architecture is able to adapt to new application code without any need to rebuild hardware. This permits a system to scale, growing in functionality and complexity, which is increasingly important for embedded systems [11]. Flexibility in terms of allowing all or part of the hardware function to be changed dynamically is also desirable, to allow custom application speed-up capability or long-term system maintenance in the field. A higher level of flexibility has a cost: reconfigurable units are physically larger than fixed units, and more hardware is dedicated to control functions. Thus, there is a tradeoff between resource efficiency and flexibility.

Architectures that use software on a general purpose CPU are moderately flexible, as the software can be changed. However, they are not necessarily as efficient as architectures that include custom hardware. Even the simplest CPUs introduce an overhead for fetching, decoding and executing instructions. ASIPs do not provide any more flexibility than a general purpose CPU as their hardware is fixed once defined, although resource efficiency may be improved by the move towards fixed units.

Run-time reconfigurable architectures (FGRAs and CGRAs) provide a greater degree of flexibility than ASIPs, as parts of the hardware can be changed dynamically. They retain the flexibility of CPUs as far as execution of new software is concerned. Resource efficiency is lost, as there are more reconfigurable units, but the flexibility and higher throughput of the device can make up for this.

1.1.4 Summary

Table 1 lists the technologies mentioned in this section, with an analysis of their throughput, predictability, relative transistor count and flexibility. An ideal architecture would provide high throughput and high predictability, with a minimal transistor count. It would also be flexible, meaning that it could easily be adapted for new tasks. No system currently meets this ideal.

Table 1. Summary of architecture types described in section 1.1.

Technology	T'put	Pred.	T. Count	Flex.
CPU	Low	High	Low	Low
CPU + Cache	High	Low	High	Low
ASIP	High	High	Low	Low
FGRA	High	Low	High	High
CGRA	High	Low	High	High
Ideal	High	High	Low	High

2 MCGREP

MCGREP (**m**icroprogrammed **c**oarse **g**rained **re**configurable **p**rocessor) is both a CPU and a reconfigurable logic device, sharing a single set of functional units arranged as a 1D array. This section examines the design principles of MCGREP, then describes its architecture and operation.

2.1 Design Principles

Heckmann *et. al.* [10] gives a list of recommendations for a CPU that is easy to model for the purposes of WCET analysis. These include separate instruction and data caches, cache replacement strategies that always lead to known states, static branch prediction, in-order execution, and no shortcuts in the hardware design. These recommendations lead to a CPU that is predictable, but not as fast as a CPU optimized for a high average speed (assuming the same silicon process for manufacture).

The use of application specific hardware can speed up performance of a CPU by many orders of magnitude [7]. Importantly for a RTS, this may be done while maintaining predictability. However, the cost is flexibility, as hardware is being committed to a fixed purpose.

Reconfigurable architectures provide a way to add application specific hardware without reducing flexibility [25]. In FGRAs reconfiguration is achieved at a high cost, due to the complexity of the reconfigurable platform and the resulting size of the configuration bitstreams. In contrast, CGRAs have lower reconfiguration costs than FGRAs [24], due to the reduced size of the required configuration bitstream, which is helpful in an RTS that must remain responsive. Some CGRAs, such as PipeRench [20], introduce a pipeline that allows the CGRA configuration to be changed every clock cycle.

Summarizing, MCGREP follows Heckmann's principles for a predictable CPU, but couples this with the flexibility of a CGRA. To further increase flexibility, the CGRA is controlled by a microprogram [18].

2.2 Architecture

MCGREP is connected to other components (RAM, ROM, devices) in a conventional manner (Fig. 1). The internal architecture of MCGREP is illustrated in Fig. 2.

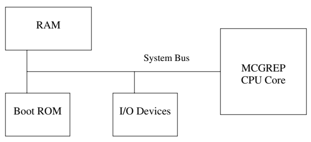

Figure 1. Bus connection block diagram for a MCGREP system.

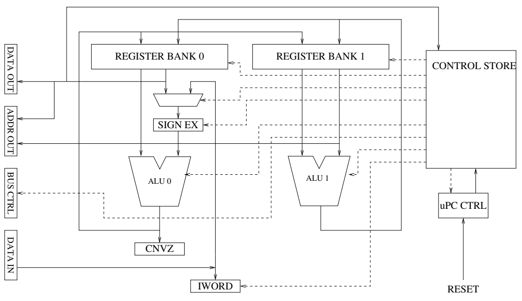

Figure 2. Current MCGREP Architecture.

MCGREP is microprogrammed, based on classic microprogramming architectures [18]. In the figure, dotted lines represent control paths, carrying instructions from the microprogram store (right of figure) to other devices. The microprogram store is a fast internal RAM with a very wide data output. Control paths are connected to one or more bits, possibly with a simple intermediate decoder to reduce the RAM required. Fig. 3 gives an example of the relationship between microcode bits and two internal CPU devices.

MCGREP provides two modes of operation, namely CPU and application specific. These are discussed in the following sections.

2.2.1 Conventional CPU Operation

On startup, the operation of MCGREP is controlled by an initial microprogram that emulates a conventional CPU,

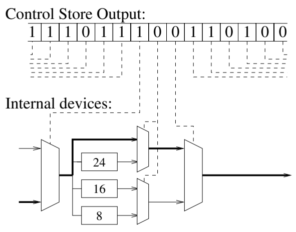

Figure 3. Example of the relationship between microcode bits and a multiplexer (left) and a sign extender (center). The bits select the path taken by the data.

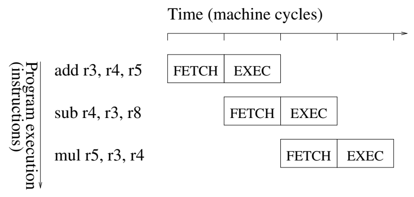

Figure 4. MCGREP's two-stage pipeline. Most opcodes execute in one machine cycle. Table 6 lists exceptions: longer opcodes stall the pipeline until they complete.

fetching program instructions from external RAM. These instructions are executed using a two-stage pipeline, illustrated in Fig. 4. This shallow pipeline does not require any hidden state components in order to operate at maximum efficiency. Instruction decode is part of the fetch cycle.

All MCGREP instructions take a fixed number of clock cycles to complete and are unaffected by execution history, making MCGREP a predictable processor. A three-stage pipeline, involving a separate decoding stage, would have similar characteristics with a lower overall propagation delay - this is planned for future versions.

2.2.2 Application Specific Operation

In conventional CPU mode, MCGREP may execute any program compiled with the correct ISA. However, execution speed is bounded by the speed of the memory, as MCGREP has no cache. To obtain maximum throughput, programs may upload new microprograms into the microprogram store, and then trigger execution using a special instruction. Essentially, this permits application specific operations to be encoded as single instructions. Uniquely, MCGREP allows new microprograms to be uploaded dynamically at run-time (unlike ASIPs); from either application or system software.

MCGREP microprograms can be viewed as sequences of configurations for a CGRA. The two execution units in the center of Fig. 2 are time-multiplexed, allowing them to act as a virtual CGRA (Fig. 5). CGRAs offer useful capabilities for parallel execution at the microinstruction level. The correct sequence of CGRA configurations can carry out any function that would normally be executed by machine code, but in fewer clock cycles. The speedup is achieved by parallelism, lack of a decoding step, and the high speed of microcode store access.

Fig. 6 shows a sequence of opcodes for a *restricted instruction set computing* (RISC) processor. These can be mapped as a sequence of three MCGREP CGRA configurations (plus an exit configuration) as shown in Fig. 7.

A program obtains a CGRA mapping through a bespoke

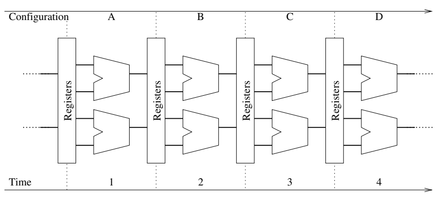

Figure 5. MCGREP's execution units, viewed as a virtual CGRA.

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

compiler that translates conventional opcodes (or explicitly specified register transfers) into microcode. Microcode is stored as read-only data in programs, and uploaded either on initialization or on demand.

There is limited space for configurations within MCGREP's internal memory. However, given that microcode configurations can be changed dynamically, an OS or task can multiplex microcode configurations as required. The finite space limits the number of functions that can be accelerated at any one time, but this is not a problem for many programs, as it is sufficient to accelerate *hotspots* - the areas of code that are most frequently executed. However, MCGREP is not limited to hotspots. It may accelerate any area of code, including worst-case execution paths.

Complete reconfiguration of the MCGREP execution units takes place every machine cycle, according to the configuration stored in microcode. This allows different speed up functions to be executed on different clock cycles without affecting the predictability of the processor. These functions can achieve a higher throughput than plain machine code, as they carry out operations in parallel, fetch coefficients from reserved areas of the register file, and require no instruction fetch or decoding step.

2.3 Related Work

Hotspots are prioritized for optimization almost universally. ASIP tool vendors recommend profiling to find hotspots [7], caches are effective because they store hotspot instructions, and *just in time* (JIT) compilers often target hotspots first. This is because hotspots have a more significant effect on overall throughput than any other part of the code. For more information about hotspots and the techniques used to detect them, the reader is referred to [16].

Reconfigurable arrays have been combined with earlier processors. [9] gives an overview. Chimaera [27] is of particular interest as the reconfigurable array and processor share a register file, a model which is reused in MCGREP in order to achieve tight integration. However, reconfiguration is not instantaneous and requires suspension of normal execution. PipeRench [20] improves on this using a time-multiplexing technique, reused by MCGREP, but the PipeRench array is not tightly integrated into the CPU. ReRisc [24] tightly integrates a RISC core and a CGRA with configuration cache, permitting fast reconfiguration in the case of a cache hit.

Run-time reconfigurable hardware is a young research field which has not been extensively applied to real-time systems. Steiger [23] describes an *operating system* (OS) for sharing an FPGA between real-time tasks, in which the FPGA is used as a fine-grained reconfigurable array. A similar approach is taken by Shang [21]. The techniques used by this work, which primarily involves the solution of a 2D scheduling problem, are specific to fine-grained arrays. CGRAs such as MCGREP may reuse software task scheduling approaches to multiplex configurations.

However, the difficulties posed by CPU hidden state have been examined extensively. Some research aims to solve the problem through better WCET analysis techniques [4, 10]. Other work reduces the complexity of the problem. The VISA approach [1] creates a simple model of a complex processor to facilitate WCET analysis, and then bounds the operation of the complex processor to the timing of the simple model. Cache locking [3] may also be used to ensure that a real-time task is always kept in cache. This is an effective approach, but complex analysis is still required for the parts of the program that are not in cache.

User-programmable microcode has been made available before, for example by the PDP-11, but the feature was rarely used, primarily because microcode is highly specific to a particular internal CPU architecture. There is no layer of abstraction between hardware and microcode, so preserving binary compatibility prevents any improvements being made to the CPU architecture.

This disadvantage is avoided by MCGREP by making the internal architecture fully open and providing tools to translate register transfers into microcode. To preserve binary compatibility, it is proposed that descriptions of microcode in an abstract form should be stored with compiled applications so that microcode can be compiled for the current processor before execution. Future work on MCGREP will introduce a "CPU driver" to make this process almost transparent to the programmer.

MCGREP's approach bears some similarities to VLIW [6], which also permits software to direct the

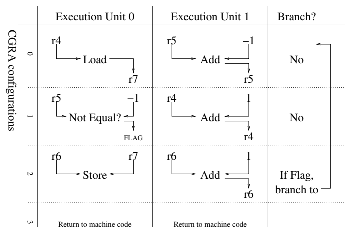

Figure 7. Mapping of Fig. 6 onto a sequence of MCGREP configurations.

low-level operations of functional units. The key difference is in the level of dynamic control of features. MCGREP allows a program to decide how to use operation storage and computing resources within the processor, which may be done according to any criteria required by the designer, statically or dynamically. In VLIW approaches, decisions about the use of computing resources must be made at compile time, and decisions about the use of operation storage resources are made by the cache hardware.

3 MCGREP Implementation

The implementation of MCGREP has the architecture illustrated in Fig. 2. It is written in VHDL as a softcore: a hardware definition that can be downloaded onto an FPGA. It includes two execution units (center of figure) as this is the minimum number required to demonstrate the principles of the design: the simplest possible implementation. Execution is directed entirely by the microcode store (right side of figure), which is a "block RAM" within the FPGA.

The 32-bit OpenRISC [12] *instruction set architecture* (ISA) was adopted for MCGREP's conventional execution mode. OpenRISC is a free softcore processor. MCGREP offers a reasonable level of compatibility with OpenRISC, supporting all instructions generated by the OpenRISC compiler.

Any ISA could be used, but the OpenRISC architecture is stable, freely available, and supported by free software tools such as `gcc`, the `glibc` library, and both RTEMS [17] and Linux operating systems. Reuse of this ISA also permits direct comparison of MCGREP with the OpenRISC softcore, as both run the same machine code. An interpreter for OpenRISC instructions is preloaded into the microcode store, to take care of booting the processor and running user programs.

MCGREP programs may be built using a conventional tool flow (C source $\rightarrow$ Object code $\rightarrow$ Executable). Programs built in this way will be compatible with both MCGREP and OpenRISC. However, MCGREP programs may also make use of application specific microcode. For this,

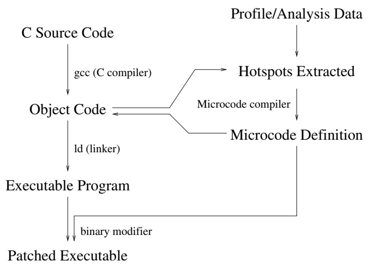

Figure 8. Conventional toolchain with support for appplication specific microcode.

```
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 r3,0xfffffefc(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 r11,0xffffff24(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 9. A hotspot from `mad`, in OpenRISC machine code form.

a different tool flow is used (Fig. 8). New steps include identification of candidate code areas for microcode implementation (this may be done by profiling or analysis), microcode generation, and binary modification. The binary modification step replaces existing machine code with a call to equivalent (but faster) microcode.

3.1 Microcode Generation

Figures 9 through 11 illustrate the microcode generation process as applied to `mad`, the MPEG audio decoder [13]. This begins with the OpenRISC machine code shown in Fig. 9, which was identified by profiling as one of several hotspots in `mad`. This machine code is translated into a *register transfer form* automatically. The register transfer form describes the operation in an abstract notation that is independent of the hardware. For example, $D \leftarrow A\ op\ B$ means that the result of the operation *op* on the data from registers A and B is stored in D.

```
00: $r4 \leftarrow r6$ shr 0x18
01: $r12 \leftarrow r4 \wedge$ 0xf
02: flag $\leftarrow r12 \neq 0$
03: $r29 \leftarrow 1$, if flag { branch to state 10 }
04: $r10 \leftarrow 0$, $[r16] \leftarrow r10$
05: $r21 \leftarrow r6$ shr 0x14, $r12 \leftarrow r21 \wedge$ 0xf
06: flag $\leftarrow r12 \neq 0$, $r5 \leftarrow 1$
07: if flag { PC $\leftarrow$ PC + 0xab60 - 0xac48, return to program }
08: PC $\leftarrow$ PC + 0xabb0 - 0xac48, return to program
09: $r23 \leftarrow 0$
10: $r3 \leftarrow [r2-$0x104$]$, $r4 \leftarrow r29$ shl $r12$
11: flag $\leftarrow r27 = 0$, $r27 \leftarrow r3 \wedge r4$
12: $r7 \leftarrow r12$ shl 2, if flag { branch to state 20 }
13: $r31 \leftarrow r7 + r2$, $r19 \leftarrow r14 - 1$
14: $r11 \leftarrow [r31-$0xdc$]$, $r17 \leftarrow 1$
15: $r15 \leftarrow r17$ shl $r19$, $r12 \leftarrow r18 \wedge r15$
16: flag $\leftarrow r12 = 0$, $r14 \leftarrow r19$
17: if flag { PC $\leftarrow$ PC + 0xacc0 - 0xac48, return to program }
18: $r11 \leftarrow 0 - r11$,
19: $[r16] \leftarrow r11$, branch to state 5
20: PC $\leftarrow$ PC + 0xacc8 - 0xac48, return to program
```

Figure 10. Optimized microprogram sequence for Fig. 9, expressed as register transfers. The sequence has been simplified slightly: load and store actually require several microinstructions. Branches that leave microcode are relative to the entry point making the microcode position independent.

```
static const unsigned microcode [] = {
    0x382424fa , 0x4a1020e0 , 0x803 ,
    0x78180dfa , 0xC5C42434L , 0x804 ,
    0x7a1300f7 , 0xa009090 , 0x60a ,
    0x391324fa , 0x4409090 , 0x80a ,
    0x382424fa , 0xa04b820 , 0x802 ,
    ...  } ;
```

Figure 11. Microcode commands embedded in C source.

After this, the register transfers are converted into an optimized sequence in which operations are parallelised whenever possible (Fig. 10). Currently, the optimization process is done by hand, but techniques for automatic optimization of register transfers are well-known and will be introduced in subsequent versions of the tool.

Finally, the optimized register transfers are automatically compiled into microcode. A C source file containing a table of microcode commands is generated, ready for uploading into the microcode store (Fig. 11). On bootup, the microcode store is preloaded with an initial configuration for conventional execution: new commands have to be loaded in the area of memory that is not used by this.

The microcode store is memory-mapped, so uploading involves writing the commands to a series of special addresses in memory. MCGREP includes a driver procedure, written in C, which will take a table like the one shown in Fig. 11 and upload it into the processor. No checking is performed on the commands.

The compiler supports both static configurations (which are fixed at program initialization) and dynamic configurations (in which the microcode is changed in response to program execution). It also acts as the core generator for the MCGREP core itself (implemented in VHDL).

Table 2. Sizes and maximum speeds of some 32-bit softcores on Virtex-II.

Soft-core	Size (LUTs)	RAM (kb)	Speed (MHz)
OR1200	5286	0	46.9
MCGREP	2591	28	44.3
Microblaze	1149	0	110.9

4 Evaluation

In this section, the tradeoffs made by MCGREP are evaluated. The experimental platform is a Xilinx Virtex-II 2000 FPGA (speed grade 4) with 512Kb of SRAM, in which MCGREP is supported by a boot ROM, I/O devices, and a variety of hardware timers that are used for measurements. Some experimental evaluations use benchmark programs taken from the MIBench [8] and Mediabench [13] suites.

Comparisons can be made between MCGREP and other softcore processors. A direct comparison is possible with the OpenRISC OR1200 [12], as MCGREP implements the subset of the OR1200 ISA used by the C compiler (`gcc`). The Microblaze [2] softcore is also compared, as it is a popular architecture for embedded systems built on Xilinx FPGA platforms.

4.1 Instruction Throughput versus Transistor Count

Transistor count represents the amount of physical hardware required by a device. As the target platform for MCGREP is an FPGA, the number of *look-up tables* (LUTs) are used in place of a transistor count, as LUTs are the smallest general-purpose unit on an FPGA. Table 2 compares the size and maximum frequency of three softcores implemented on a Virtex-II 2000. All cores were configured with near-identical features where possible (no cache, no FPU, no MMU, and multiplier), and synthesized using identical settings. The latest CVS version of OpenRISC OR1200 was used, along with Microblaze version 3. All cores use identical peripherals via Wishbone on-chip buses. The amount of RAM used by MCGREP is dependent upon the amount of microcode store required. The range is currently from 4.5kb to 28kb.

Table 3 shows the instruction throughput of some benchmarks on each processor, when running in a predictable (*i.e.* cacheless) configuration. Each throughput value is calculated by dividing the total number of instructions executed for the benchmark by the execution time of the benchmark. Variations are caused by a different mix of instructions in each benchmark.

Table 3. Instruction throughputs of benchmarks on three processors with 40MHz clock and single-cycle memory latency (millions of instructions per second).

B'mark	O'RISC	M'blaze	MCGREP mc only	MCGREP mc+μc
`aes`	9.99	10.49	7.09	11.09
`crc32`	11.43	11.42	8.89	21.78
`dijkstra`	14.57	11.18	12.06	18.52
`g721`	15.20	11.91	13.04	17.04
`jpeg`	10.26	9.88	8.09	11.56
`mad`	17.81	9.89	13.56	19.99
`qsort`	8.46	11.07	8.87	11.68
`sha`	14.51	11.38	11.95	26.42

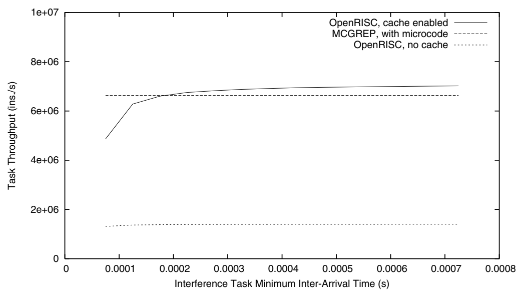

Figure 12. The effects of interference on throughput of the `crc32` benchmark, on the OpenRISC and MCGREP processors. OpenRISC results were obtained with and without caching.

Table 4. Number of instructions executed for each benchmark.

Benchmark	OR/MCGREP instructions	Microblaze instructions
`aes`	1.91×10^6	1.86×10^6
`crc32`	3.94×10^6	3.94×10^6
`dijkstra`	2.81×10^8	1.73×10^8
`g721`	3.37×10^8	2.78×10^8
`jpeg`	7.36×10^6	5.53×10^6
`mad`	5.58×10^7	3.34×10^7
`qsort`	5.01×10^6	5.18×10^6
`sha`	7.01×10^7	5.50×10^7

For reference, Table 4 shows the number of instructions executed by each benchmark, when compiled for OpenRISC/MCGREP and Microblaze. Data sets are identical - ISA and compiler differences account for the variable numerical relationship of the instruction counts. These results show that the Microblaze compiler is slightly more efficient than the OpenRISC compiler in some cases, with the same program requiring fewer instructions.

Table 3 shows two sets of results for MCGREP - one when using plain machine code only, and another using a combination of machine code and microcode. The effect of the microcode is identical to the effect of the original machine code, so the total number of instructions executed is considered to be the same.

Table 3 indicates that MCGREP's performance is slightly poorer than OpenRISC and Microblaze when executing machine code ("mc") only. When application specific microcode is in use ("mc+μc"), MCGREP's overall performance exceeds that of both OpenRISC and Microblaze in all of the cases tested here. Thus, MCGREP compares well with OpenRISC and Microblaze: it requires less general-purpose hardware than OpenRISC, and only around twice that required by Microblaze.

4.2 Instruction Throughput versus Predictability

An RTS designer is interested in proving timing correctness. MCGREP attempts to facilitate CPU modeling [10] and measurement [4] approaches to WCET analysis by operating in a simple and highly predictable fashion.

The execution time of a task T should be independent of the operation of all other tasks on the system. However, other tasks may affect the execution time of T through the CPU cache and other hidden state elements. In particular, higher priority tasks and interrupt handlers may preempt T at any time, causing hidden state information (*e.g.* cached instructions) to be lost. This reduces the throughput of T - the portion of the total throughput of the system that is specific to the execution of T. This effect is shown in Fig. 12.

The hidden state in a cache has no effect unless memory latency is significant, so the test system used to generate the following results has a memory latency of 25 CPU clock cycles. This simulates a typical ratio between memory and CPU speed. Due to the limitations of the FPGA platform, the frequency of each CPU is 40MHz, but the results are equally applicable at any higher frequency.

Interference is generated by a sporadic interference task that runs with a time interval in the range $[t, 2t]$. The minimum value of t was set to 2000 clock cycles (50μs at 40MHz) - simulating a fast interrupt. The interference task is a short routine that invalidates the instruction cache.

Table 5 shows the effects of interference on various benchmark tasks running on MCGREP, OpenRISC, and Microblaze, with minimum and maximum throughputs. OpenRISC and Microblaze make use of 4kb instruction caches (more than enough to contain the hotspots from each

Table 6. Instruction timings on MCGREP.

Instruction class	Example	Clock cycles
ALU	`l.add`	max(2, m)
Shift	`l.sll`	max(2, m)
Shift Imm.	`l.slli`	max(4, m)
Multiply	`l.muli`	max(2, m)
Load	`l.lwz`	4 + 2 × max(2, m)
Store	`l.sw`	2 + 2 × max(2, m)
Branch	`l.bnf`	max(2, m)
Special	`l.mfspr`	max(4, m)
Set Flag	`l.sfeqi`	max(2, m)
Move High	`l.movhi`	max(2, m)

benchmark). The throughputs are measured during execution of the benchmark task only: scheduler overheads and the interference task are not counted.

Table 5 shows large variations in execution times for the processors with caches. For example, when the `aes` benchmark executes on OpenRISC, the worst case throughput is 1.06, and the best case throughput is 3.63. Thus, the WCET of an `aes` task could vary by a factor of nearly 3.5 times. On MCGREP, the `aes` throughput varies by (at most) a factor of 1.03 (due to a small inaccuracy introduced by software control of a counter). The processors with caches do achieve higher peak throughput than this version of MCGREP, but MCGREP can achieve the same level of throughput under any amount of interference.

4.3 Timing Analysis

To demonstrate MCGREP's support for easy timing analysis, Table 6 gives the number of clock cycles required by a number of operations. In this table, m is the memory latency in clock cycles. Instruction fetch also requires m clock cycles, but fetch is carried out in parallel to execution (see Fig. 4), so m is a minimum bound on each timing, not an additional cost. Using Table 6, a static analysis tool can derive exact timings for sections of MCGREP code. Application specific microcode timings are found by using the rule that each microcode state requires 2 clock cycles per execution, plus memory latency for loads and stores.

The hotspot shown in Fig. 9 can be analyzed for WCET using these rules. The hotspot is broken down into basic blocks (Fig. 13): the time taken for each is computed (Table 7), and the longest possible path through the execution graph gives the WCET. The longest path (1, 5, 6, 7, 8, 3, 4) requires 82 clock cycles when the hotspot is executed as machine code (assuming $m = 0$), and 46 clock cycles when executed as microcode as shown in Fig. 10.

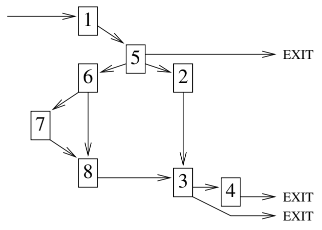

Figure 13. Execution graph for the basic blocks in the hotspot from Fig. 9.

Table 7. Execution times for the basic blocks in the hotspot from Fig. 9. Times are given in clock cycles with the assumption that $m = 0$.

Basic block	Address range	μcode states	mc ET	μc ET
1	[ac48,ac58]	[0,3]	12	8
2	[ac5c,ac60]	4	8	6
3	[ac64,ac74]	[5,7]	12	6
4	[ac78,ac7c]	8	4	2
5	[ac80,ac94]	[9,11]	20	12
6	[ac98,acb8]	[12,17]	24	10
7	acbc	18	2	2
8	[acc0,acc4]	19	8	6

4.4 Flexibility versus Resource Efficiency

Table 2 showed that MCGREP is around half the size of OpenRISC when configured with similar features. This is because of the heavy use of FPGA RAM resources, which are used in place of general purpose FPGA fabric. Therefore, MCGREP makes more efficient use of the general purpose FPGA area, at the cost of RAM. However, this choice enables dynamic reconfiguration, bringing high flexibility.

MCGREP is a more flexible platform than OR1200 and Microblaze, as it includes a user-programmable reconfigurable logic device (CGRA) in addition to its capability for executing software. The CGRA may be programmed on a per-application, per-task, or per-function basis according to user requirements. There is no limit to the number of configurations that may be stored in memory. Loading time is predictable and constant for any particular configuration.

5 Using MCGREP Flexibility for RTS Runtime Support

In MCGREP, application specific microcode can replace any sequence of machine instructions. The most obvious use of this feature is accelerating applications. However, it

Table 5. Minimum and maximum throughput for various benchmarks on three processors, at 40MHz, with 25 cycle memory latency. Caches are invalidates every 50μs (for minimum) or never (for maximum). Values are millions of instructions per second.

Benchmark	OpenRISC	Microblaze	MCGREP mc only	MCGREP mc+μc
`aes`	$[1.06, 3.63]$	$[1.31, 4.41]$	$[1.00, 1.03]$	$[2.88, 2.91]$
`crc32`	$[4.86, 7.15]$	$[5.66, 7.05]$	$[1.15, 1.19]$	$[6.63, 6.63]$
`dijkstra`	$[3.74, 8.84]$	$[4.13, 6.22]$	$[1.57, 1.61]$	$[4.26, 4.38]$
`g721`	$[1.59, 3.92]$	$[2.71, 4.63]$	$[1.20, 1.24]$	$[1.99, 2.04]$
`mad`	$[1.52, 3.71]$	$[1.59, 3.10]$	$[1.33, 1.37]$	$[2.27, 2.31]$
`qsort`	$[1.58, 6.33]$	$[2.89, 7.31]$	$[1.13, 1.27]$	$[1.69, 1.77]$
`sha`	$[3.45, 6.74]$	$[4.53, 7.48]$	$[1.15, 1.18]$	$[4.71, 4.78]$

may also assist an RTS in a number of other ways, described in the following section.

5.1 Accelerating System Code

MCGREP permits frequently active operating system features, such as context switchers, schedulers and interrupt handlers, to be partially moved into microcode. Even microcode loaders can be resident in microcode. The use of microcode within the operating system would reduce the system overhead, increasing the response time of application tasks. Fig. 7 is an example of system code (`memcpy`, from the C library) translated into an optimal microprogram. Any function calling `memcpy` will benefit from the increased speed.

5.2 Smart Interrupt Handling

A typical CPU forces interrupt handling code to run at a higher priority than all other code, with the exception of code that executes while interrupts are disabled. An RTS may include many tasks at different priority levels, but even the highest priority task can still be preempted by a signal for a low priority task. This results in a part of of the low priority task (the *interrupt service routine* or ISR) getting exclusive control of the system while the interrupt is handled. The conventional solution to this problem is to introduce *two-level interrupts* in which the ISR is as short as possible, but generates an event that causes the low priority task to be scheduled once higher priority tasks complete.

In MCGREP, the first part of this two-level interrupt scheme can be implemented directly in microcode, which eliminates any need to save registers, context switch, or fetch instructions from memory. MCGREP interrupts are handled during instruction decode: when an interrupt is pending, the instruction decoder will reroute microprogram execution to a handler routine. The present version of MCGREP uses this feature to emulate OpenRISC-style interrupt handling. However, this behavior may be extended.

```
00: $IN \leftarrow$ [0x80000000]
01: $IP \leftarrow [IN + TD1]$
02: flag $\leftarrow IP > TP$
03: if flag { branch to state 8 }
04: $X \leftarrow [IN + TD2]$
05: $X \leftarrow X + 1$
06: $[IN + TD2] \leftarrow X$
07: return to program
08: trigger software ISR (PC $\leftarrow ISR$)
```

Figure 14. Microcoded ISR for a two-level interrupt scheme. As in Fig. 10, load/store operations have been simplified for clarity.

Fig. 14 shows an alternative microcoded ISR. This ISR acknowledges an interrupt (by reading the interrupt number IN from a memory mapped device at address 0x80000000). It then obtains the priority for the interrupt IP, by loading entry IN from a priority table in memory starting at $TD1$. This interrupt priority is compared to the current task priority TP, stored by the OS during the last context switch. If $IP \leq TP$, the interrupt is marked pending by incrementing a counter in a pending table (starting at $TD2$). The ISR will be invoked by the scheduler at a later stage. However, if $IP > TP$, the interrupt service routine is invoked immediately.

With this scheme, each interrupt has a (dynamic) priority, and immediate interruption is only permitted if this priority exceeds the priority of the current task.

As earlier descriptions of microcode have indicated, there is no limit on the complexity of the commands that could be executed by an ISR. MCGREP is flexible enough to support any OS signaling protocol and may adapt to new protocols without any need to change the hardware.

5.3 Test and Set

An *atomic operation* cannot be split or interrupted. It is guaranteed to complete. Atomic operations are commonly

```
00: $X \leftarrow [rA]$
01: flag $\leftarrow X \neq 0, [rA] \leftarrow rB$
02: return to program
```

Figure 15. Microcoded test-and-set instruction. The value at rA is loaded and copied into the flag register, then rB is stored at rA. These operations are indivisible.

used to implement *protected objects* and *monitors* - high level objects in which mutually exclusive access is guaranteed. CPUs often support mutual exclusion using a "test and set" instruction [22], which will read and update a control flag atomically. This is used as the core of a software function to manage mutual exclusion (such as `pthread_mutex_lock` from POSIX threads).

In MCGREP, microcode execution is always atomic, so microcode is a suitable platform for implementing mutual exclusion functions. Interrupts are only handled during instruction decoding, which takes place only during a "return to program" microcode operation. The main benefit of microcoding mutual exclusion operations is execution speed, as no extra steps need to be taken to make the operations atomic. Flexibility is still assured as microcode operations can be changed as easily as software. Fig. 15 gives an example of a test-and-set instruction for MCGREP.

5.4 Priority Inheritance

It is possible to implement more sophisticated types of mutual exclusion than test and set in hardware, but this is rarely done because such schemes are inflexible and may result in hardware that an OS will not be able to use due to incompatibility with its own architecture. However, MCGREP provides a way to implement complex mutual exclusion schemes at the microcode level, which can be easily changed to match OS and application requirements.

For example, semaphore locking is commonly used to protect access to a critical section, and some critical sections are shared between tasks of different priority. In these cases, *priority inversion* may occur (Fig. 16(a)). In this figure, inversion occurs because low-priority task L locks a critical section at point A, but access to the section is later required by high-priority task H at point B. As task M has priority over L, H is blocked by M as it waits for L to complete. *Priority inheritance* avoids this by giving L the priority of H at point C (Fig. 16(b)).

MCGREP's microcode may be used to implement a *priority ceiling* protocol with minimal context switching. This priority inheritance scheme prevents priority inversion. The immediate priority ceiling protocol [5] (ICPP) is used.

Critical section entry is handled by the microcode in Fig. 17, and exit is handled by Fig. 18. If the CPU flag is set on return from the exit microcode, the scheduler should be

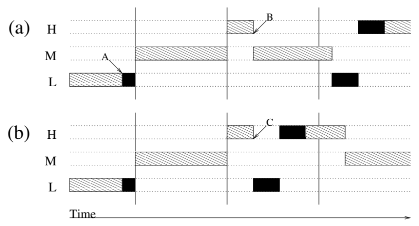

Figure 16. A priority inversion problem, (a), is solved using priority inheritance, (b). Critical sections are marked in black.

Microcode:

```
00: flag $\leftarrow CP \geq TP$
01: if not flag, { return to program }
02: $CPS \leftarrow CPS + 4, TP \leftarrow CP$
03: $[CPS] \leftarrow CP$
04: $[PV] \leftarrow CP$
05: return to program
```

Pseudocode:

```
if ( $CP \geq TP$ )
then
  – Priority is updated. Push:
  $CPS := CPS + 4$ ;
  $[CPS] := CP$ ;
  – Update stored priority
  $TP := CP$ ;
  $[PV] := CP$ ;
end if ;
```

Figure 17. Critical section entry.

invoked by a "yield" operation as task priorities have been updated. For these examples, the current task is known as T and the critical section is known as C. The registers that are used are listed in Table 8.

5.5 Instant Context Switching

MCGREP's architecture also provides many more registers than are required by RISC code. These registers are currently used to store temporary data and constants (*e.g.* the microcode in Fig. 10 uses about 15 immediate values which are stored as constants in the register file). They may also be used for fast context switching, by partitioning the register file and swapping to another partition on task switch. This technique is fast, but inflexible as the maximum number of possible partitions is fixed.

6 Conclusions

This paper has described MCGREP, a processor architecture combining coarse-grained dynamically reconfig-

Microcode:

```
00: flag $\leftarrow TP = CP$
01: if not flag, { return to program }
02: $CPS \leftarrow CPS - 4$
03: $X \leftarrow [CPS]$
04: flag $\leftarrow X \neq TP$
05: if not flag, { return to program }
06: $TP \leftarrow X, [PV] \leftarrow X$
07: return to program
```

Pseudocode:

```
if ( $TP = CP$ )
then
  – Pop
  $CPS := CPS - 4$ ;
  $X := [CPS]$ ;
  if ( $X \neq TP$ )
  then
    $TP := X$ ;
    $[PV] := X$ ;
    – Program should call yield ()
  end if ;
end if ;
```

Figure 18. Critical section exit.

Table 8. Key to register names used in Figures 17 and 18.

Register	Description
OTP	Static task priority for T.
TP	Dynamic task priority for T.
CP	Static ceiling priority for C.
CPS	Pointer to top of stack containing the ceiling priorities of critical sections that are locked. Initially contains OTP. Stack memory is local to T.
PV	Pointer to the global variable that stores the priority of T. This is read by the scheduler.
X	Temporary store.

urable logic with conventional processing features. Real-time systems design presents many architectural choices, and MCGREP adds a new option: a predictable processor that achieves speed through support for application specific microprograms which direct the operation of the reconfigurable logic.

MCGREP has been implemented on an FPGA, and compares well to existing softcores. Some of the uses for MCGREP in real-time applications have been demonstrated by experiment. In particular, MCGREP tasks are unaffected by the operations of higher-priority tasks, as MCGREP lacks any hidden state features (such as caches).

References

[1] 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 '03*, pages 350–361, 2003.

[2] Anonymous. Embedded processing and control solutions for Spartan-3 FPGAs. Application Note XAPP477, Xilinx Corporation, 2003.

[3] A. Arnaud and I. Puaut. Dynamic Instruction Cache Locking in Hard Real-Time Systems. In *Proc. of the 14th International Conference on Real-Time and Network Systems (RNTS)*, Poitiers, France, May 2006.

[4] G. Bernat, A. Colin, and S. M. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. In *Proc. RTSS*, pages 279–288, 2002.

[5] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, 2001.

[6] 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.

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

[8] 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.

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

[10] 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.

[11] International Technology Roadmap for Semiconductors. ITRS Design 2005. `http://www.itrs.net/Common/2005ITRS/Design2005.pdf` (accessed 18 Apr 06).

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

[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] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Proc. RTSS*, page 12. IEEE Computer Society, 1999.

[15] C. Maxfield. *The Design Warrior's Guide to FPGAs*. Academic Press, Inc., Orlando, FL, USA, 2004.

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

[17] OAR Corporation. RTEMS (accessed 16 Jan 06). `http://www.rtems.com/`.

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

[19] M. Reshadi and D. Gajski. A cycle-accurate compilation algorithm for custom pipelined datapaths. In *Proc. CODES/ISSS*, pages 21–26, 2005.

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

[21] L. Shang and N. K. Jha. Hardware-Software Co-Synthesis of Low Power Real-Time Distributed Embedded Systems with Dynamically Reconfigurable FPGAs. In *Proc. ASP-DAC*, page 345, 2002.

[22] A. Silberschatz, P. Galvin, and G. Gagne. *Applied Operating System Concepts*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[23] C. Steiger. Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks. *IEEE Trans. Comput.*, 53(11):1393–1407, 2004. Herbert Walder and Marco Platzner.

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

[25] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte. The Molen Polymorphic Processor. *IEEE Trans. Comput.*, pages 1363– 1375, November 2004.

[26] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Proc. Int. Conf. Quality Software*, Sep. 2005.

[27] 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.