Boosting Single Thread Performance in Mobile Processors using Reconfigurable Acceleration

Geoffrey Ndu

School of Computer Science

University of Manchester

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences
2012

Contents

Thesis Word Count: 39397

List of Figures

List of Tables

Listings

Abstract

Boosting Single Thread Performance in Mobile Processors using Reconfigurable Acceleration
Geoffrey Ndu
A thesis submitted to the University of Manchester for the degree of Doctor of Philosophy, 2012.

Mobile processors, a subclass of embedded processors, are increasingly employing CMP (Chip Multi-Processor) designs to improve performance. Single-thread performance in CMP suffers as vendors move to fewer per core resources to enable them to instantiate more cores on a die. Single thread performance is still important according to Amdahl's law. The traditional technique for efficiently boosting serial performance in embedded processors, dedicated hardware acceleration, is unsuitable for modern mobile processors because of the heterogeneity and the diversity of applications they run. This thesis investigates the possibility and potential benefits of using a general purpose accelerator (placed within the datapath of a CPU), reconfigured on an application-by-application basis, as a means of efficiently increasing single-thread performance. Configurations for the accelerator are generated at runtime, via a JIT compiler, which allows it to accelerate dynamically generated code. This is important as dynamic code generation is now prevalent on mobile processors.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the thesis, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487), in any relevant Thesis restriction declarations deposited in the University Library, The University Library's regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The University's policy on presentation of Theses

I would like to dedicate this thesis to my loving family.

Acknowledgements

Thanks to Dr. James (Jim) Garside, my supervisor, and to the whole APT group for their help and encouragement. A big thank you to Dr. Andrew Bardsely and Dr. Plana Cabrera, my proof readers, for their fast turn-around times.

Special thanks to Imepras for donating software tools used in this project.

1

Introduction

Mobile processors, a subclass of embedded processors, are General Purpose Processors (GPPs) designed primarily for small, fan-less, battery powered, mobile computing devices such as smart-phones. They are characterised by high performance, low energy consumption, small area and low cost. Mobile processors began as processors tuned for particular classes of applications, Application Specific Instruction-set Processor (ASIP). For instance, early mobile phones had baseband processors (e.g. Philips KISS-16-V2 [1]). Simple scalar processors augmented with Application Specific Integrated Circuits (ASICs) for processing speech and radio signals in real-time. Baseband processors also run the user interface on phones but at a lower priority. The dedicated hardware accelerators on these processors improved performance and reduced energy consumption in a cost and area effective manner [1].

Mobile processors have evolved from ASIPs to full blown GPPs as mobile computing moved from embedded devices, with few functionalities, into portable, connected, general purpose computers. A typical Android [2] smart-phone can install and run about 500 000 [3] diverse applications, often called 'apps', ranging from Linpack [4], a library for numerical linear algebra, to Visidon AppLock [5], a face recognition tool for protecting applications. Relying on dedicated accelerators, as in the early days of mobile phones, to boost performance is no longer practical as apps are diverse and not known at design time. Further, mobile processors are increasingly being used outside portable devices such as in micro-servers. Micro-servers are servers built using a large number of relatively slow

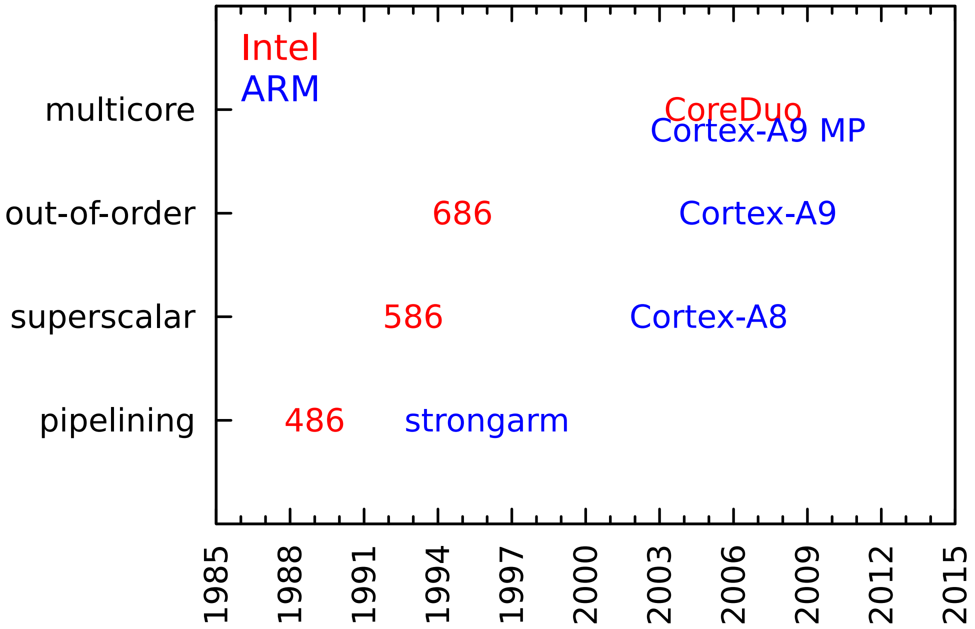


Figure 1.1: Intel (desktop) vs. ARM (mobile).

but cheap and power efficient mobile processors [6]. Therefore, a modern mobile processor is more like a conventional (desktop) processor, see figure 1.1, making it a challenge to manage the contradictory constraints of power/energy, area, flexibility and cost.

Mobile processor vendors are increasingly employing Chip-MultiProcessor (CMP) designs, multiple 'simple' Central Processing Units (CPUs) (often called 'cores') on the same integrated circuit die, to improve the performance and energy efficiency instead of more powerful single processors. Single processors are no longer scaling in performance (see Figure 1.2 for past and future trends in microprocessor technology), because technology scaling no longer provides consequent clock frequency improvements. Furthermore, it costs less to exploit the ever increasing number of transistors provided by process technology by simply instantiating more copies of the relatively simpler (and relatively easier to design) CPU in each successive chip generation instead of designing and debugging a new large single processor each time. A mobile CMP is actually part of a System-on-Chip (SoC) that enables a typical mobile device such as a smart-phone

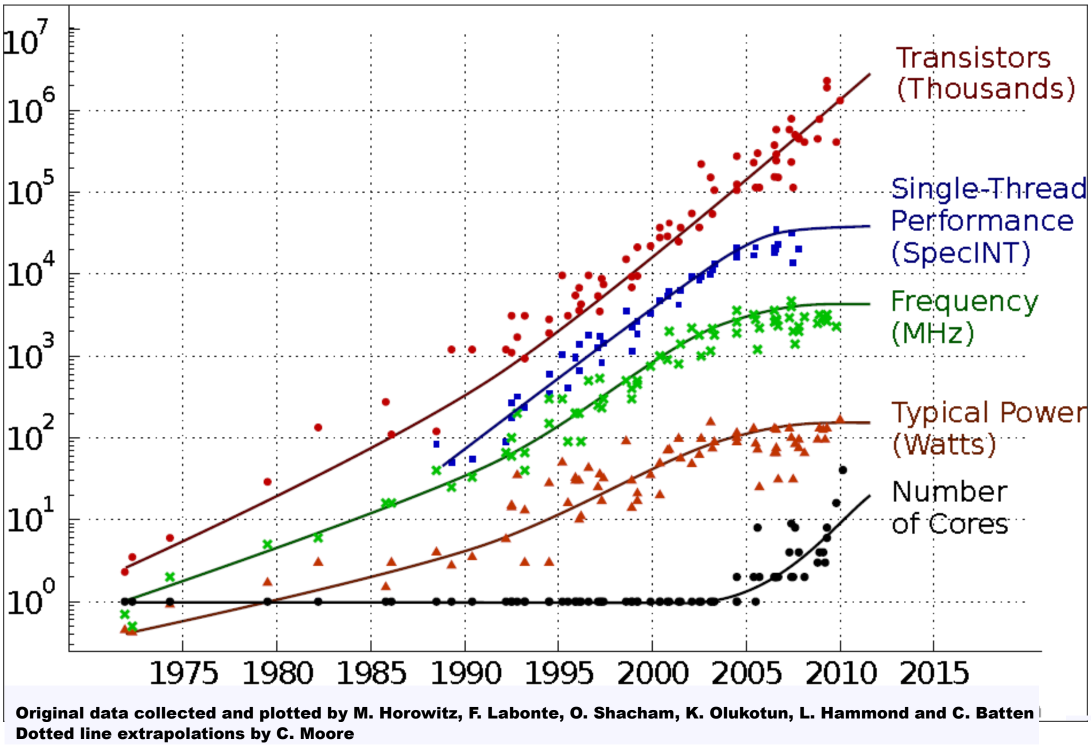


Figure 1.2: Microprocessor trends: past and future [7].

to provide more processing power than was available to supercomputers of the 1970s at 4×10^{-11} of the cost; see Table 1.1. For instance, the Texas Instrument® OMAP5432 SoC has up to 12 processors in a 14 mm × 14 mm package that runs within the maximum system power limitation of 2.5–3.0 W for mobile devices while costing less than $50. These processors range from a 2-core 2 GHz mobile processor to a 16-core General Purpose Graphics Processing Unit (GPGPU), see Figure 1.3, all running off a 1000–1500 mAh battery.

CMPs, also called multi-core microprocessors, improve performance and reduce power consumption by handling more work in parallel on simpler, more energy efficient CPUs compared to single processors. They exploit Thread-Level Parallelism (TLP) [8], distributing execution processes/threads across cores [8, 9]. Performance improvements depend on the fraction of the application that is parallelised. Performance is compromised for lowly threaded applications as CMPs tend to have limited single thread capabilities (because of fewer per core resources).

Single thread performance is still important as some key applications have limited TLP. According to Amdahl's law [10], serial sections within a parallel application are performance constraints. Current and future challenge for mobile processor vendors is how to increase single thread performance in these resource-constrained, general purpose cores efficiently. Most traditional methods employed by conventional cores are unsuitable for mobile processors because of energy, area and cost issues.

Table 1.1: Supercomputer Cray-1 vs Mobile Phone HTC HD2.

Feature	Cray-1	HTC HD2
Year	1976 [11]	2009 [12]
Number of processors	1 [11]	1 [12]
Frequency(MHz)	80 [11]	1000 [12]
MFLOPS(Linpack 100)	3.4 [13]	74 [14]
Volume(mm^2)	≈196 846 795 [11]	88 845 [12]
Power(W)	115000 [11]	3 [12]
Cooling	Freon [11]	none [12]
Weight(kg)	4762 [11]	0.2 [12]
Price($ Million 2009)	20–36 [15]	0.0008
Main Memory(MB)	8 [15]	512 [12]

1.1 Improving Single Thread Performance

Improvements in single thread performance, at the architectural level, have largely come from superscalar processing and pipelining. Superscalar processing [16] exploits Instruction Level Parallelism (ILP) [17, 18] by dynamically increasing the number of instructions issued simultaneously to functional units on the processor. Pipelining [19] breaks down a single instruction into a sequence of small steps and performs the steps of adjacent instructions in parallel, allowing the clock rate to be increased more than suggested by Moore's 'law' [20]. Unfortunately, extracting

ILP from an instruction stream is power-hungry as the complexity of the additional logic needed for dynamic discovery of ILP is approximately proportional to the square of the number of instructions that can be issued simultaneously [9]. Cost and power issues mean that mobile processors do not employ 'aggressive' superscalar processing. Further, the degree of parallelism in a typical instruction stream is limited [21], so using transistors to build even more complex superscalar processors achieves very little additional benefit for most applications.

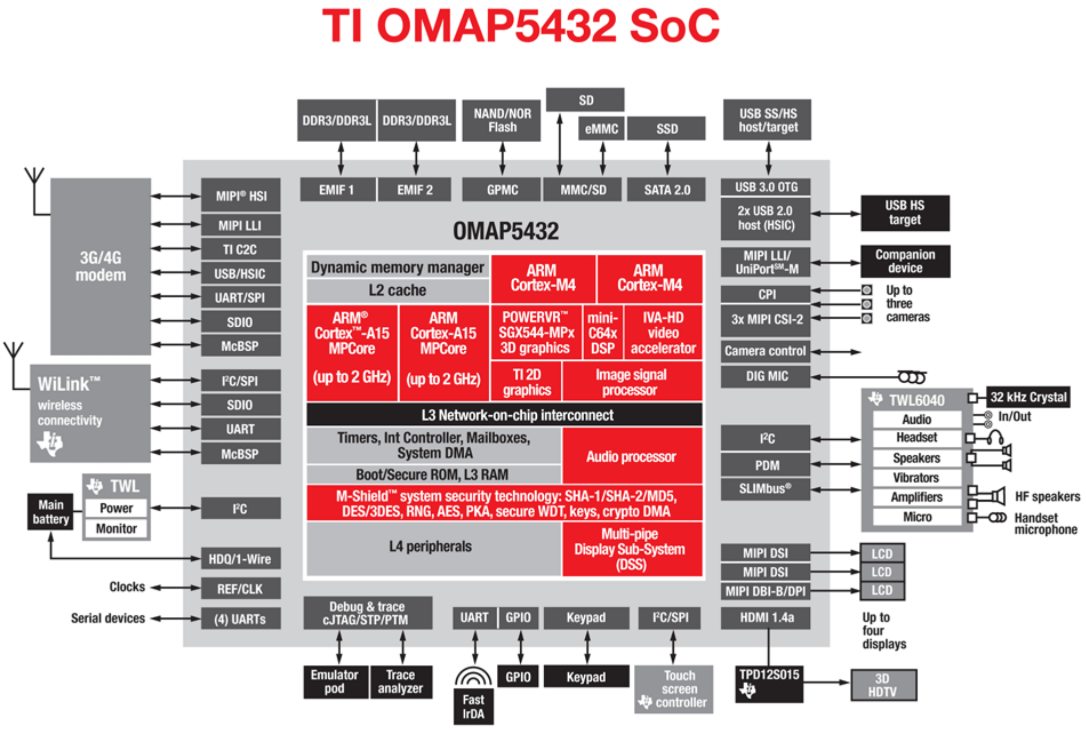


Figure 1.3: A state-of-the-art mobile SoC. Courtesy of Texas Instruments.

Similarly, increasing performance via deeper pipelines is expensive as more transistors are needed for adding pipeline registers and bypass path multiplexers, increasing power consumption further. This, and performance losses from pipeline flushes, primarily caused by branches, combine to make very deep pipelining unsuitable for mobile processors.

The time-tested approach of accelerating compute-intensive parts of application using dedicated hardware is not suitable for mobile processors because of the heterogeneity and diversity of applications they run. The next best alternative is having ‘general purpose’ Reconfigurable Hardware (RH), which can be reconfigured on an application by application basis to implement frequently occurring functions. This approach, reconfigurable computing, is less efficient in terms of area, cost and power than fixed hardware but allows a GPP to be *specialised based on the application it's currently running*. GPPs augmented with RH are known as Reconfigurable Processors (RPs). RH has been used successfully to accelerate single thread performance in experimental and commercial processors [22, 23, 24, 25, 26]; the challenge is *how to map dynamically generated code onto the reconfigurable hardware efficiently*.

1.2 The Need for Dynamically Generated Code

Software cost is now significant, up to 80% of the development cost for mobile computers [27]. This, and the short development cycles of mobile computing devices makes it inconceivable to rewrite millions of lines of code for each processor generation. Some mobile ‘ecosystems’, such as Google's Android [2], require all but low-level system code to be written in platform independent Intermediate Representation (IR) which is compiled to binary ‘just-before-use’ via a Virtual Machine (VM). Mobile processors are designed to be binary compatible across generations and, at most, a simple recompilation is all that is required to port software to a new generation. With a mobile CMP, simply moving a parallel application to a newer generation of processor may degrade performance as the application may be very sensitive to hardware parameters such as core count or inter-core latencies, that vary from one generation to another. Consequently, mobile software development is gradually moving to programming systems that support forward-scaling [28].

Forward-scaling is the ability of parallel application performance to scale with new core counts and cope with constant evolution of the Instruction Set Architecture (ISA) with little or no rewriting and compilation i.e. binary and performance portability [28]. For example, the data width of vector extensions increase with each new generation of chip causing difficulties in terms of binary compatibility. Forward-scaling systems such as Array Building Blocks (ArBB) [28] generate vector instructions at runtime to match the width on the processor even though the programmer can't know what they might be when the program was created.

One of the key enablers of forward-scaling in programming systems is dynamic compilation [29, 30]—the runtime compilation of platform independent IR to native code—which allows applications to adapt to particular architectural characteristics at runtime. Most state-of-the-art parallel programming systems now support dynamic compilation. This, potentially, allows applications to take advantage of the latest hardware and software features without the need for re-compilation at the cost of runtime overheads.

Dynamic software compilation is now so common on mobile processors that vendors provide hardware to speed it up. For instance ARM processors have ThumbEE and Jazelle [31]. Most state-of-the-art Placement and Routing (P&R) only support off-line mapping on the RH which is inadequate for mobile computers where compilation is now largely dynamic. They need the compiler or programmer to identify suitable section(s) for acceleration and to synthesize configurations for them off-line. The dynamic generation of configuration, dynamic mapping, is particularly difficult for RPs as time consuming tasks such as P&R now need to be performed at runtime. Recent studies [22, 23, 24, 32, 33] have investigated dynamic mapping to RH using Dynamic Binary Translation (DBT) [34]. A dynamic binary translator, which could be software, hardware or a software/hardware hybrid, is used to map sequences of machine code to the RH. DBT for RH has the advantage of low overheads since the semantic gap between microprocessor machine code and RH configurations is small compared to translating from a high-level language. Binaries, however, lack the high-level semantic information required for efficient mapping to the RH requiring binary translators to decompile binaries to recover such information [23]. The sophistication of decompilation differs from design to design. Decompilation is expensive and does not always recover enough high level information. Furthermore, DBT introduces additional overheads reducing performance and increasing energy consumption. Even without the drawbacks of DBT, present DBT-based dynamic mappers are still unsuitable for mobile CMPs because of area, cost, resource (e.g. memory footprint) or power issues.

This thesis advocates dynamic mapping based on compilation from IR instead of DBT. RH dynamic compilers can be integrated into VMs already present on the CMP eliminating the need for an extra, distinct translation layer. Furthermore, IRs usually contain enough high level information to remove the need for decompilation. The translation overheads, however, are high compared to DBT. This thesis explores a hardware/software co-design approach to reducing the overheads of dynamic compilation on RH.

1.3 Contributions

This thesis explores the possibilities and potential benefits of an architecture that uses RH to accelerate single thread execution of dynamically generated code on mobile CMPs. Its main contribution is the co-design of a mobile CMP (augmented with RH) and associated RH dynamic compiler the Just-In-Time (JIT) variant capable of accelerating single threads in the code produced at run-time. It specifically targets forward-scaling programming systems such as the Array Building Blocks where kernel (critical) code is dynamically generated.

Specifically, the key contributions include:

- VIrtual REconfigurable Micro-ENgine for Translation (VIREMENT), a mobile CMP architecture where each CPU has reconfigurable hardware integrated as a functional unit. The architecture is restrictive which reduces mapping overhead enabling the run time generation of configurations. The reconfigurable hardware speeds up single threads by exploiting the ‘small’ ILP within a typical basic block. As such, it consists of a two-dimensional array of interconnected simple, programmable, word-sized functional units with data routed using multiplexers.

  Since CMPs have stringent cost constraints (e.g. chip size), the design aims to keep the reconfigurable extension to the CPU small by using a reconfigurable array of modest size.

- Dynamic Compilation Engine (DCE), a JIT compiler for VIREMENT which dynamically translates IR into configurations for the RH. The DCE design aims to be a back end JIT complier for the numerous VMs available on various mobile software platforms but the primary targets are VMs of forward-scaling programming systems. Like hardware, mobile software platforms have stringent cost constraints (e.g. memory footprint) as such DCE design places emphasis on being lean and fast. The JIT is intended for the translation of kernels (critical sections of an application) to enhance the chances of amortization of overheads.

P&R in the DCE is with a novel, single pass algorithm suitable for resource constrained mobile platforms. The algorithm was developed as part of this thesis.

1.4 Overview

The rest of this thesis is divided into seven chapters:

Chapter 2 reviews reconfigurable processing in general with emphasis on the challenges of dynamic translation. It also discusses related work.

Chapter 3 presents a quantitative characterization of the targeted application and how that impacts the design of VIREMENT.

Chapter 4 discusses the design of VIREMENT.

Chapter 5 discusses the design of DCE and illustrates its operation with some examples.

Chapter 6 presents the proposed evaluation methodology and framework for VIREMENT and DCE.

Chapter 7 is an evaluation of the VIREMENT architecture using the the methodology and framework proposed in chapter 6.

Chapter 8 concludes the thesis with a summary, a list of achievements, and an outline of potential improvements to VIREMENT.

1.5 Publications

The following articles, based on the work presented in this thesis, have been published or accepted for publication:

1. Geoffrey Ndu, Jim Garside: **Boosting Single Thread Performance in Mobile Processors via Reconfigurable Acceleration**. 8th International Symposium on Applied Reconfigurable Computing (ARC 2012): 114-125

2. Geoffrey Ndu, Jim Garside: **Architecture for Runtime Hardware Compilation**. 5th HiPEAC Workshop on Reconfigurable Computing 2011, and also as: HiPEAC Technical Report, TR-HiPEAC-0014, January 2011

3. Geoffrey Ndu, Jim Garside: **A Co-designed Dynamic Compilation Framework for Reconfigurable Processors** (Abstract). Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011), March 2011.

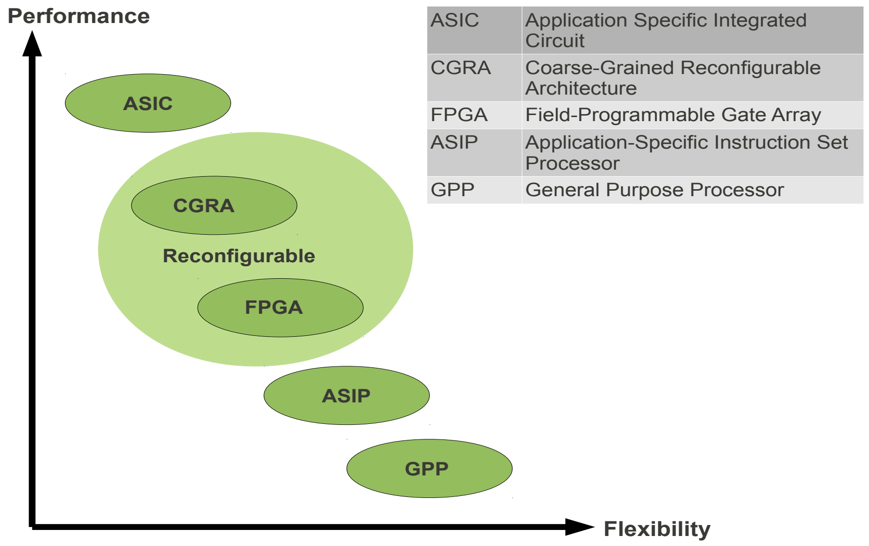
2
Reconfigurable Computing

Reconfigurable computing employs some form of post-fabrication programmable hardware potentially allowing the same hardware to be specialised to fit any application. A Reconfigurable Hardware (RH) combines post-fabrication programmability with the spatial (parallel) computation style of ASICs, unlike the less efficient temporal (sequential) computation style of Von Neuman processors [35]. Reconfigurable computing offers a good balance between the contradictory design aims of implementation efficiency and flexibility. This trade-off is depicted in Figure 2.1 for common processing architectures. From the figure, flexibility improves at the expense of efficiency. Augmenting a General Purpose Processor (GPP) core with RH potentially improves efficiency at the cost of reduced flexibility.

2.1 Reconfigurable Computing Advantages

Implementing a section of an application using RH may speed up performance significantly compared to software on a microprocessor, especially if the section is highly parallel e.g. extensive bit-level manipulation.

The efficiency of bit-level operations can be improved significantly by using an Field-Programmable Gate Array (FPGA), a type of RH, instead of a microprocessor. Bit-level operations are inefficient on microprocessors as they typically require a separate instruction or several instructions. Figure 2.2 compares bit reversal on an FPGA to a microprocessor. Figure 2.2a is an efficient software implementation of a bit reversal [36] that requires approximately 38 instructions and 48 cycles



ASIC	Application Specific Integrated Circuit
CGRA	Coarse-Grained Reconfigurable Architecture
FPGA	Field-Programmable Gate Array
ASIP	Application-Specific Instruction Set Processor
GPP	General Purpose Processor

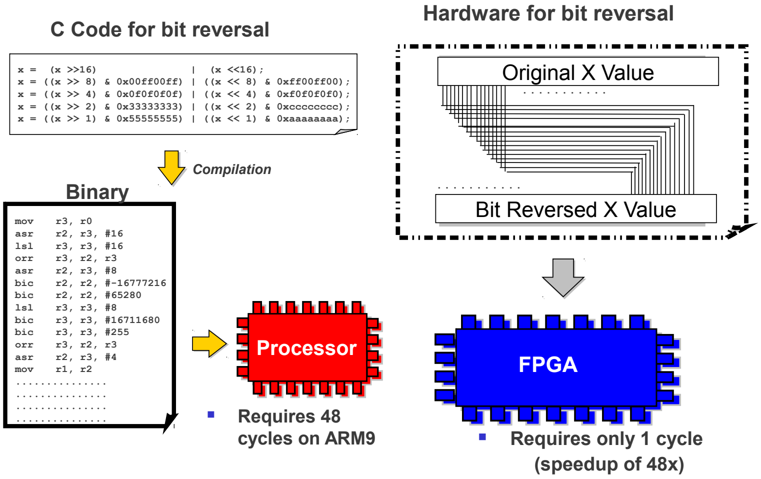
Figure 2.1: Flexibility and performance trade-off for various processing architectures

to reverse a 32-bit integer on ARM9Erev2 . However, the same operation can be performed in a single cycle on an FPGA, by simply reversing the connections betweens two buffers as shown in figure 2.2b, achieving a speedup of 48× (assuming equal cycle lengths).In this example, an instruction to achieve the same effect has been added to later implementations of ARM which illustrates that efficiency in ‘rarely’ used operations may be important. However, adding many specific instructions to a GPP is not practical because of the limited hardware budget and cost.

2.2 Reconfigurable Hardware

2.2.1 Granularity

The RH design space is extensive and can be broadly classified into two segments—fine-grained and coarse-grained—based on the data width of the smallest Processing Element (PE). Fine-grained reconfigurable hardware, like an FPGA, employ PEs of data width 1 i.e. bit-level programmable. Conversely, coarse-grained hardware


C Code for bit reversal
Compilation
Binary
Processor
Requires 48 cycles on ARM9
Hardware for bit reversal
Original X Value
Bit Reversed X Value
FPGA
Requires only 1 cycle (speedup of 48x)

```
x =  (x >>16)                | (x <<16);
x = ((x >> 8) & 0x00ff00ff) | ((x << 8) & 0xff00ff00);
x = ((x >> 4) & 0x0f0f0f0f) | ((x << 4) & 0xf0f0f0f0);
x = ((x >> 2) & 0x33333333) | ((x << 2) & 0xcccccccc);
x = ((x >> 1) & 0x55555555) | ((x << 1) & 0xaaaaaaaa);
```

```
mov    r3, r0
asr    r2, r3, #16
lsl    r3, r3, #16
orr    r3, r2, r3
asr    r2, r3, #8
bic    r2, r2, #-16777216
bic    r2, r2, #65280
lsl    r3, r3, #8
bic    r3, r3, #16711680
bic    r3, r3, #255
orr    r3, r2, r3
asr    r2, r3, #4
mov    r1, r2
...............
...............
...............
...............
```

(a) bit reversal ARM9Erev2. (b) bit-reversal on FPGA.

Figure 2.2: Comparing bit-reversal on ARM9Erev2 and FPGA.

employs PEs working at word level. Fine-grained hardware is very flexible and can implement arbitrary digital logic. However, its flexibility leads to performance, area and power inefficiencies as will be explained shortly. Therefore, a typical FPGA compared to standard cell ASIC requires approximately 20 to 35 times more area with a speed roughly 3 to 4 times slower and consumes roughly 10 times as much dynamic power [37]. The huge difference in efficiency is better understood by looking at the architecture of a typical FPGA, the Altera Cyclone II [38].

At the heart of the Cyclone II PE is the Lookup Table (LUT). This is a high-speed 16 × 1 Static Random-Access Memory (SRAM). The PE is programmed by loading a function's truth table. Therefore, any logic function with one output and up to four inputs can be implemented on the LUT. More complicated functions can be implemented by aggregating several LUTs.

The LUT and other logic such as flip-flops make up a Logical Element (LE) and 10 of them form a Logic Array Block (LAB). Figure 2.3 shows a simplified LE. Cyclone II devices range in capacity from 4,608 to 68,416 LEs interconnected in a two-dimensional row and column-based architecture. Generally, routing is unsegmented but each wiring segment spans only one LAB before terminating in a switch box. Consequently, an FPGA requires a lot switches, typically 200–400 per

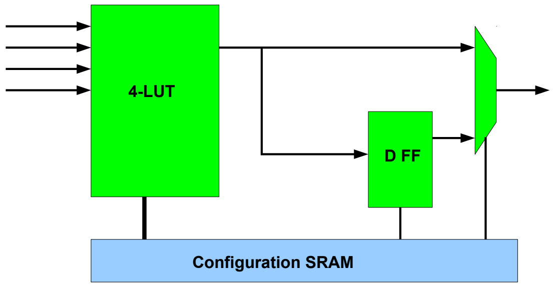


Figure 2.3: Simplified depiction of an LE [40].

4-LUT resulting in an inefficient implementation, especially when used to compose operators for wide datapaths. FPGAs also require large configuration memories because of the high number of configuration points. Partial reconfiguration—reprogramming a portion of the FPGA while the rest is still operating—could be used to reduce/hide reconfiguration times at the cost of increased hardware and software complexity.

Some of the disadvantages of FPGAs, lack of forward compatibility and long reconfiguration times, can be tackled with Pipeline Reconfigurable FPGAs (PipeRench architecture) [39]. PipeRench breaks large, single static configurations into pieces that correspond to pipeline stages in the application.These small configurations are swapped in and out automatically at run-time allowing a circuit that would otherwise be too large to fit on the available hardware. Configurations compiled for one instance of PipeRench can used for a larger (or smaller) instance of PipeRench with a corresponding increase (or decrease) in performance.

In summary; fine-grained hardware trades area, speed, and power for flexibility. As such, they are unsuitable for designs where power efficiency is of primary importance, and with their long configuration time it would be slow to implement an execution model which relies of a rapid change of configuration.

In contrast to fine-grained hardware, the PEs in a coarse-grained hardware are complex logic blocks ranging from Arithmetic and Logic Units (ALUs) to small Reduced Instruction Set Computer (RISC) type programmable cores often arranged as a 2D array. The use of complex logic blocks allows for an efficient implementation of complex operators in silicon rather than having to compose such operators from LUTs. Coarse-grained hardware also has fewer configuration points leading to a significant reduction in configuration data and time. However, they are potentially inefficient for bit-level processing. There are also unable to leverage optimizations in the size of operands as the size of each logic block is static. For instance, configuring a 64-bit ALU for 8-bit operations leads to a high degree of inefficiency.

2.2.2 Reconfiguration Model

The reconfiguration model is largely determined by when a new configuration can be loaded into the reconfigurable hardware. Traditionally, a new configuration is only loaded at the beginning of execution and remains unchanged for the duration of the application. This is referred to as static reconfiguration and requires the system to be halted while the reconfiguration is in progress.

Dynamic reconfiguration allows concurrent reconfiguration and execution. This is achieved by grouping configurations into contexts allowing the device to quickly switch between different planes, or context to be swapped as needed. Code that may not have been able to implement completely onto the reconfigurable can be partitioned and swapped as needed.

The dynamic reconfiguration model enables implementation which are otherwise too large for the RH. Reconfiguration overhead, however, becomes significant if not carefully managed [41].

2.2.3 System Architecture

It is often impractical to implement the entirety of an application on a RH. Most applications have large number of sections that are executed relatively infrequently, and attempting to implement all of these sections on the RH would be non-viable (a section needs to be executed sufficiently frequently on the RH to recover configuration overhead). In practice, RH is used to accelerate only the most critical sections (kernels). As such, the RH is usually coupled to a host processor(s) that executes the non-critical control sections. However, some reconfigurable systems [42, 43] are without host microprocessors.

The RH and its host can be integrated in a number of ways as shown in figure 2.4. In some systems [44, 45, 46], the RH is a separate device attached to the

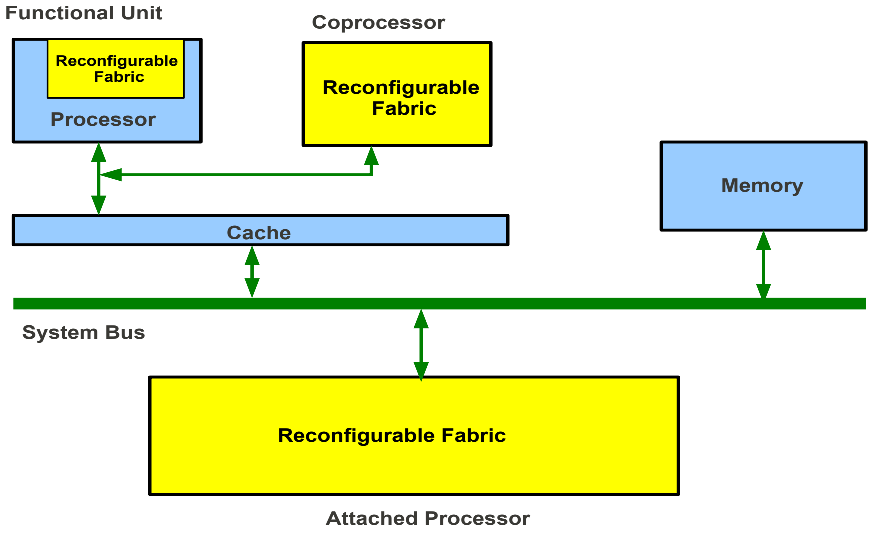


Figure 2.4: Types of RFU coupling

host processor through the system bus. The host processor simply offloads kernels to the hardware for processing. However, the overheads of communicating over the system bus makes this approach suitable only for computations requiring little communication between the RH and the host. For example: graphics processing, where the RH may only need the source and destination addresses from the host to process a large chunk of pixels. Systems employ SRAM-based data buffers (which consumes a significant amount of power), analogous to a data cache, between the reconfigurable hardware and the memory to service applications with data.

An attached processor system is easy to construct and does not require any modification to the host or its compiler. However, the cost of communicating over the system bus limits it to applications where the communication to computation ratio of an application is low.

The reconfigurable hardware can also be coupled to the host processor as a coprocessor [23, 47, 48, 49, 50]. This reduces communications overhead compared to an attached processor as data is exchanged using protocols similar to those used in floating point coprocessors. The host processor requires little or no modification but the communication overhead can still be significant if the communication to computation ratio is high. In such a situation, employing the RH as a extra functional unit in the host processor's data path is more suitable

[50, 51, 52]. This reduces the communication overhead to a minimum as the RH has direct access to the host processor's register file and the decoder issues 'special' instruction to the RH to perform processing just like any other functional unit. However in this case, the host processor needs to be modified to integrate the RH, often called the Reconfigurable Functional Unit (RFU). Such designs are often termed Reconfigurable Instruction Set Processorss (RISPs) [53]. The merits of these different approaches to integration are summarised in table 2.1

Coupling	Power[a]	Speed [b]	Bottleneck	Application[c]
Attached	high	slow	high overhead of system bus communication	large data set, low communication to computation ratio
Coprocessor	low	fast	limited coprocessor register file size	medium/small data set, high communication to computation ratio
Functional unit	low	very fast	limited processor register file size	small data set, very high communication to computation ratio

[a] power consumed by data-storage elements used for communication.
[b] communication speed between RH and host.
[c] characteristics of application best suited for.

Table 2.1: Comparing RH integration techniques.

2.3 Reconfigurable Architecture Programming

RPs are not yet widespread—despite significant performance improvements reported by academic and commercial projects—because they are significantly more difficult to program compared to conventional processors.

Programming a RP often involves first choosing a target processor and then writing high-level application code for it, almost always in C/C++. Coarse-grain parallelism is often required to be written in a particular style to exploit the capabilities of the architecture. The application is then optimised for the specific architecture by identifying parts of the application that can benefit from hardware acceleration, and replacing them with RH configuration(s). Compilation tools are now able to generate the RH configurations—with little or no help from the programmer—and interface them with the rest of the code. Designers, however, still prefer hand-mapping as automatic compilation remains significantly less efficient [54].

Reconfigurable computing is a volatile and fragmented field, new architectures are introduced and previous architectures retired within a short time by different vendors. These architectures do not have common hardware architecture, even those from the same vendor, thus requiring the programmer to re-program substantially each time the application is moved to a new device. This makes reconfigurable processing uncompetitive against standard microprocessors where different hardware architecture implementations are hidden behind a standardised ISA.

Dynamic mapping avoids these problems by translating code to run on RH automatically at runtime. The most common technique is translating executing microprocessor binary onto a restrictive RH (a form of DBT [34]). Restricting the flexibility of the RH drastically reduces the time and resources required to perform tasks such as P&R. Translating from binaries instead of source code keeps overheads small but requires expensive, and not always effective, decompilation to recover high-level constructs such as loops, arrays and functions which are critical to efficient mapping [23].

Mapping without decompilation might result in slower and bigger circuits—Vahid et alia [55] reported an average slowdown of 4×—while mapping with very limited decompilation lead to implementations that are only slightly faster than software [23, 55]. Therefore, decompilation is a prerequisite for effective binary-based dynamic mapping. As an example, consider that for many applications, compilation exposes parallelism by unrolling loops. Without recovering enough information about loop structures and bounds from binary, the parallelism visible to a binary translator is very limited.

The main problem with decompilation is that the representation of data and instructions in the Von Neumann architecture are indistinguishable. Data is often located in between instructions e.g. indexed jump tables. Furthermore, programmers often write self-modifying code. As such, it is hard and expensive to decompile a software binary [56]. The successes of the reconfigurable architectures that employ extensive decompilation, such as Warp [23, 55] (discussed later), is that they operate in the embedded and DSP domains, where applications tend to be written using constructs that are ideal for decompilation [57].

A Directly Interpretable Representation (DIR) [58], an IR with a simple syntax and a relatively small set of simple operators, such as Lower Level Virtual Machine (LLVM) [59], is a middle ground between the need for high level information and minimising translation overheads. It is portable across RPs and has relatively little run-time translation overhead. A DIR retains enough semantic information to allow more aggressive program transformations than are easily attainable with binary (even with decompilation).

Architectures that employ dynamic mapping are discussed next with emphasis on their suitability for mobile processors.

2.4 Warp

Warp [23] is a family of processors that automatically extracts and compiles critical software kernels to FPGA. A typical Warp processor (see figure 2.5) is an SoC with a main processor for executing application, a less powerful CAD processor on which a lean FPGA compiler—Riverside On-Chip CAD (ROCCAD)—runs, a profiler and an FPGA.

Execution starts on the main processor. A hardware profiler monitors the executing application and determines the critical kernels defined as loops that account for 10% or more of total application execution time. The profiling result is then passed to ROCCAD which first analyses the profiling results to determine the candidates for implementation on the FPGA. The selected kernels are first decompiled to an IR and then synthesised for the FPGA. The hardware configuration generated by ROCCAD is then loaded onto the FPGA. Finally, the executing binary is patched so that the next time the kernels are encountered they are processed on the FPGA instead of the main processor.

Despite the significant reduction in execution times and energy consumption for applications, Warp still suffers from the disadvantages of FPGA-based sys-

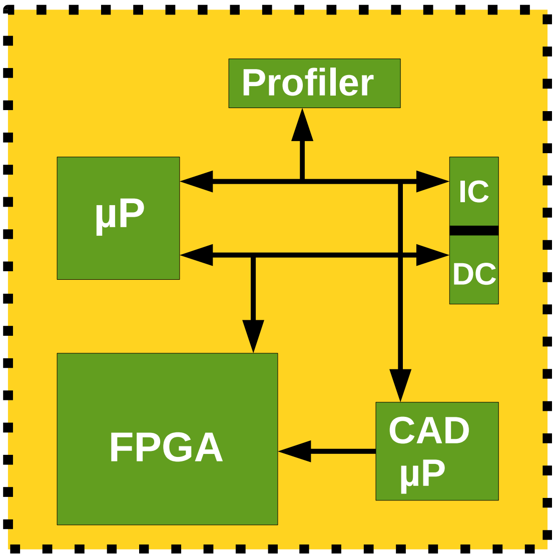


Figure 2.5: Architecture of Warp processors

tems (see subsection 2.2.2). Furthermoremore, decompilation, used to convert the executing binary to a format suitable for synthesis, is expensive and does not always recover enough high level information to generate efficient hardware. Finally, the significant on-die memory requirements of ROCCAD and the large memory footprint (which makes them unsuitable for mobile processors) for storing FPGA configurations limits Warp to only applications where a few inner loops dominate.

2.5 Configurable Compute Array

The Configurable Compute Array (CCA) [22, 60] is a matrix of simple, heterogeneous functional units, coupled to a host processor as a functional unit. The CCA, depicted in figure 2.6, is configured with hardware primitives ,'microops', similar to microcode. Data flows from top to bottom with the outputs of functional units fully connected to the inputs of the units in the next row.

Accelerating applications on the CCA involves two steps: the discovery and delineation of critical subgraphs, subsets of an application's Directed Flow Graph (DFG), suitable for the CCA and the replacement of such subgraphs with microops that configure the CCA. Static and dynamic approaches for generating microops

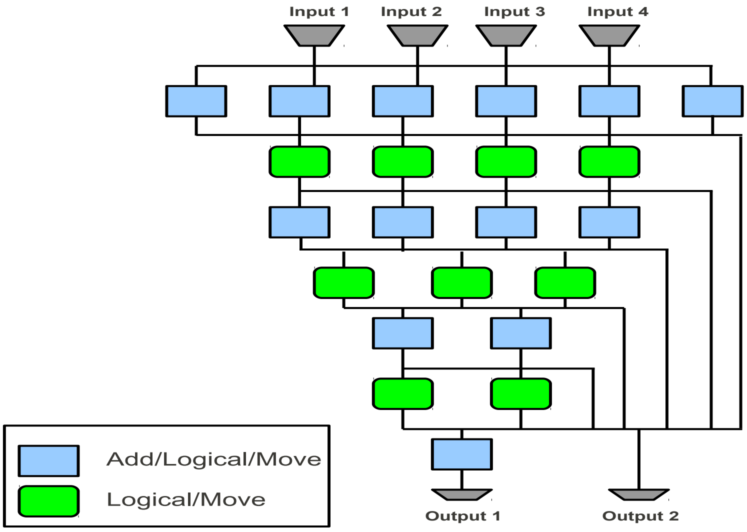


Figure 2.6: A block diagram of a depth 7 CCA.

have been presented [22, 60].

The dynamic approach uses an offline compiler to identify and delineate critical subgraphs that can be mapped onto the RH. However, mapping itself is performed at runtime by a hardware engine which generates the RH configurations necessary to execute the subgraphs. This involves initially executing critical subgraphs without the RH, constructing traces of retiring instructions using a trace cache [61] (a special instruction cache which captures dynamic instruction sequences) and feeding the traces to the mapping engine.

The dynamic mapping approach is completely transparent but requires a trace cache which is rare in a mobile processor because of area, cost and energy issues. Furthermoremore, the mapping engine, based on rePlay [62], uses a very complex graph analysis technique which leads to huge resource overhead. The static approach is similar to the dynamic one but is completely offline.

As implied by figure 2.6, suitable subgraphs are restricted to those having at most four inputs and two outputs which may be limit performance [24]. Furthermore, subgraphs must not contain memory or shift instructions, which constitute a significant proportion of operations (see chapter 3), as they are not supported on the CCA.

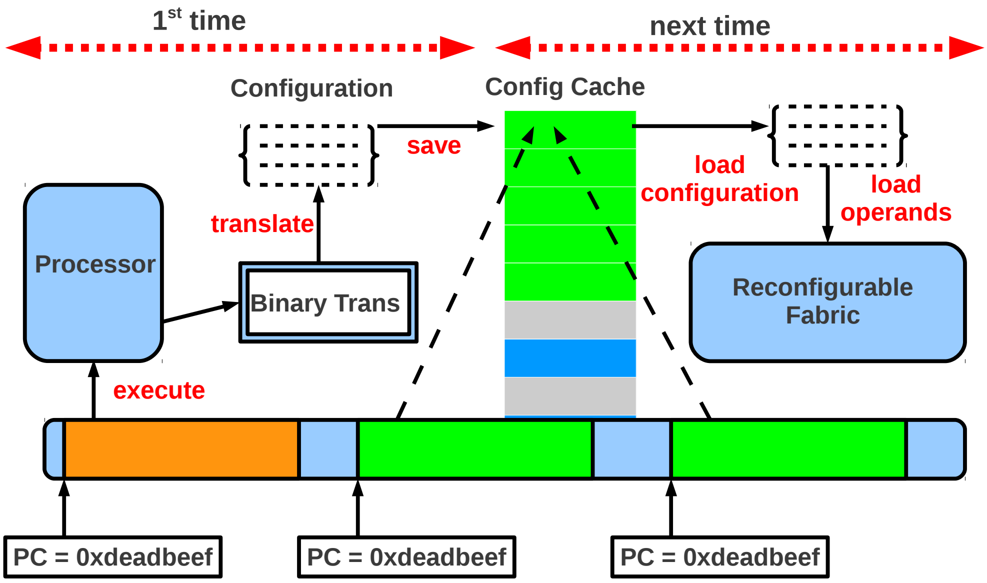


Figure 2.7: Architecture of DIM.

2.6 Dynamic Instruction Merging

Dynamic Instruction Merging (DIM) [24] (see figure 2.7) dynamically translates executing instructions onto a 2D reconfigurable array of functional units attached to a host processor as an extra functional unit. The translation algorithm, implemented in hardware, detects and transforms instruction groups into microops for execution on the RH.

Translations occur as soon as instructions are fetched and the result is stored in a dedicated configuration cache indexed by the Program Counter (PC). The next time the processor starts fetching the same sequence of instructions the processor loads the previously stored configuration from the cache and the operands from the register bank, then it activates the RFU. The RH now performs the processing and updates the PC, to the end of the sequence, for execution to continue with standard instructions.

DIM makes the RH transparent to executing binaries and can handle dynamically generated code. However, DIM takes up a very significant amount of die area. Furthermore, the binary translation algorithm is simple and fast (as it is implemented in hardware) but misses opportunities for optimising the microops produced. It has been shown that decomposing instructions into microops produces suboptimal internal code sequences [63]. Optimising the internal code sequence is non-trivial and best performed in software instead of hardware [64]. Furthermore, energy is spent translating instruction sequences that contribute little to performance as DIM attempts to translate all instructions.

Custom Reconfigurable Arrays for Multiprocessor System (CReAMS) [33] is the multiprocessor variant of DIM but has a 4-stage pipelined translation unit termed Dynamic Detection Hardware (DDH). As with DIM, the DDH is responsible for detecting and translating instructions. The DDH and its configuration memory are tightly coupled to a 5-stage processor. The host CPU, the RH and the DDH together form the Dynamic Adaptive Processor (DAP). Multiple DAPs were coupled together to form a CMP. CReAMS, like DIM, suffers from high area overhead.

2.7 RHU and RIG cores

Another project [25, 32] is based on a heterogeneous multiprocessor where each core is either a Reconfigurable Hardware Unit (RHU) or a Reconfiguration Instruction Generation (RIG). A RHU is a superscalar processor augmented with RH. The RIG is based on the same processor as the RHU but with dedicated hardware for reconfigurable configuration generation and without the RH (see figure 2.8). Each RIG services a number of RHUs.

Each RHU collects traces of committed instructions which are dispatched to a RIG for translation. A trace starts whenever a backward branch instruction commits. When a configuration is received from the RIG it is stored at the executing thread address space. When the start of a translated trace is detected it is processed using the RH instead of the standard datapath. Standard execution continues from the original instruction after the end of the translated trace. The RH is a 2D array of functional units that only handles integer ALU and simple memory operations.

A RIG generates a configuration using a combination of hardware and software. Analysis is in hardware with code generation in software that is part of the Operating System (OS).

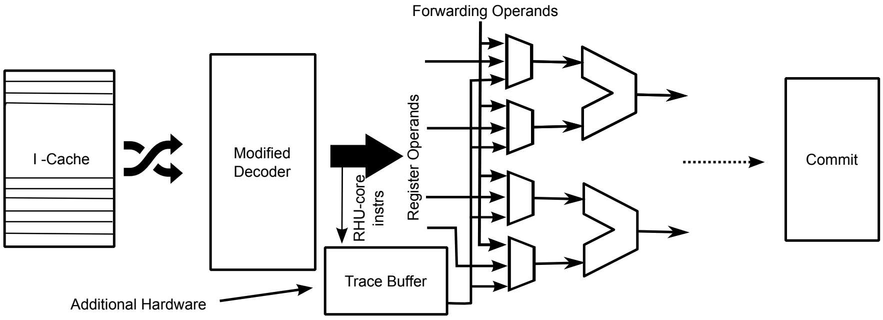


Figure 2.8: Datapath of a RIG-core.

A 4-core model, a single RIG servicing three RHU, improved per-core performance by an average of 23% [32]. However, the architecture may be too 'complex' for a mobile processor as it uses trace caches which are rare in mobile processors because of area, cost and energy issues.

2.8 Summary

This chapter introduced the concept of reconfigurable computing and discussed its advantages and potential issues . Difficulty of programming and diminished binary portability were identified as major barriers to the widespread adoption of RPs. Then, architectures that use dynamic mapping to overcome these drawbacks were reviewed. The merits and demerits of each architecture were discussed. The next chapter is on the quantitative characterization of applications targeting mobile processors with the aim of using the results to guide the design of VIREMENT.

3

Application Analysis

3.1 Introduction

Application characterisation, the quantitative understanding of workload characteristics, is important to the design of processors. This often involves instrumenting and executing reference applications to generate records and summaries of the time cost of subroutines, algorithms, number of calls etc. The output of application characterisation guides architecture and microarchiecture design.

This chapter is about the characterisation of reference applications, assembled from the Parsec [65], Rodinia [66] and Bots [67] benchmark suites. These applications are described in section 3.2 and summarised in table 3.1. The emphasis is on emerging applications such data mining. Section 3.4 describes how the insights gathered influences the design of VIREMENT.

3.2 Application/Kernel Pool

backprop Back Propagation [69] is a machine-learning algorithm, implemented in OpenMP, that trains the weights of connecting nodes on a layered neural network. `backprop`'s domain is pattern recognition and is included in this pool because of the increasing importance of pattern recognition applications—such as handwriting recognition—in mobile computing devices.

Application	Suite	Domain	Thread Imp.
backprop	Rodinia	Pattern Recognition	OpenMP
bfs	Rodinia	Graph Algorithms	OpenMP
bodytrack	Parsec	Computer Vision	Pthreads
facesim	Parsec	Physics Simulation	Pthreads
fib	Bots	Integer	OpenMP
fft	Bots	Spectral Transforms	OpenMP
fluidanimate	Parsec	Physics Simulation	TBB
freqmine	Parsec	Data Mining	OpenMp
kmeans	Rodinia	Data Mining	OpenMP
nqueens	Bots	Search	OpenMP
pathfinder	Rodinia	Grid Traversal	OpenMP
sort	Bots	Integer Sorting	OpenMP
srad	Rodinia	Image Processing	OpenMP
streamcluster	Parsec	Data Mining	TBB
vips	Parsec	Media Processing	Pthreads
x264	Parsec	Media Processing	Pthreads

Table 3.1: Application Pool (TBB refers to Intel Threading Building Blocks [68]).

bfs Breadth-First Search transverses a graph beginning at the root node and then exploring all the neighbouring nodes. Breadth-First Search is used in solving many problems in graph theory.

bodytrack tracks a human body with multiple cameras through an image sequence. The increasing significance of computer vision algorithms on mobile computing devices, where it is used in character animation and computer interfaces, leads to the inclusion of `bodytrack`.

facesim computes a visually realistic animation of a modelled face by simulating the underlying physics [70]. Computer animations increasingly employ physical simulation to create more realistic effects.

fft computes the one-dimensional Fast Fourier Transform of a vector of n complex values using the divide and conquer Cooley-Tukey [71] algorithm. It was included to represent DSP applications.

fib computes the nth Fibonacci number using recursive paralellization [67]. It was included to represent recursive algorithms that need to exploit irregular parallelism.

fluidanimate employs an extension of the Smoothed Particle Hydrodynamics method to simulate an incompressible fluid for interactive animation purposes [72]. It was included in the application pool because of the increasing use of physics simulations for animations.

freqmine identifies sets of items that often occur together in a given database using the Frequent Pattern-growth method [73]. It was included because of the emerging field of data mining in resource-constrained mobile computing environments [74].

kmeans k-means is a clustering algorithm used in data mining that aims to partition n observations into k clusters. Each observation belongs to the cluster with the nearest mean. It is used in mobile phone gesture recognition systems [75]

nqueens computes all solutions of the n-queens problem (finding a placement for `n` queens on an `n`×`n` chessboard such that none of the queens attack any other) using a backtracking search algorithm.

pathfinder uses dynamic programming to find a path on a two-dimensional grid from the bottom row to the top row with the smallest accumulated weights i.e. finds the shortest path. It is included because of the importance of mapping applications that find directions between physical locations by finding the shortest path.

sort is a parallel variation of mergesort [76]. It sorts a random permutation of `n` 32-bit numbers and was included because sorting is a computational building block of fundamental importance.

srad Speckle Reducing Anisotropic Diffusion (SRAD) is an algorithm based on partial differential equations for removing the speckles in ultrasound images without sacrificing important image features [77]. It is included in the pool because of the emergence of inexpensive mobile ultrasound using an ultrasound probe connected to a mobile phone [78].

streamcluster is a kernel that finds a pre-determined number of medians in data streams so that each point is assigned to its nearest centre as it is streamed in [65]. It is part of the application pool because of the increasing importance of data mining algorithms on mobile devices.

vips is designed for processing images larger than the amount of RAM available on a device [79]. Photo editing applications on mobile-devices are becoming more prevalent.

x264 is an application for encoding video streams into the H.264/MPEG-4 AVC format [80]. H.264, a standard for video compression, is one of the most commonly used formats for the recording, compression and distribution of high definition video.

3.3 Application Characterisation

The applications described in section 3.2 are characterised using a combination of publicly available tools (Intel Software Development Emulator [81], Microarchitecture-Independent Characterization of Applications (MICA) [82]) and custom Pin [83] based tools running on an x86 platform. Pin is a dynamic instrumentation tool for programs. It allows the injection of C or C++ code at arbitrary places in the executable. All instrumentations occur at runtime obviating the need to rewrite binaries.

A major pitfall of application characterisation is that processor microarchitecture could hide underlying, inherent program behaviour [82], hence only microarchitecture-independent application characteristics are measured. The results are presented in sections 3.3.1 to 3.3.6.

3.3.1 Dynamic Instruction Mix

The dynamic instruction mix characterises applications according to the frequency of the different instruction types encountered during execution and serves as a guide to the type of operations to support on the RH. Instructions are classified functionally into 9 groups: `load`, `store`, `branch`, `integer arithmetic`,

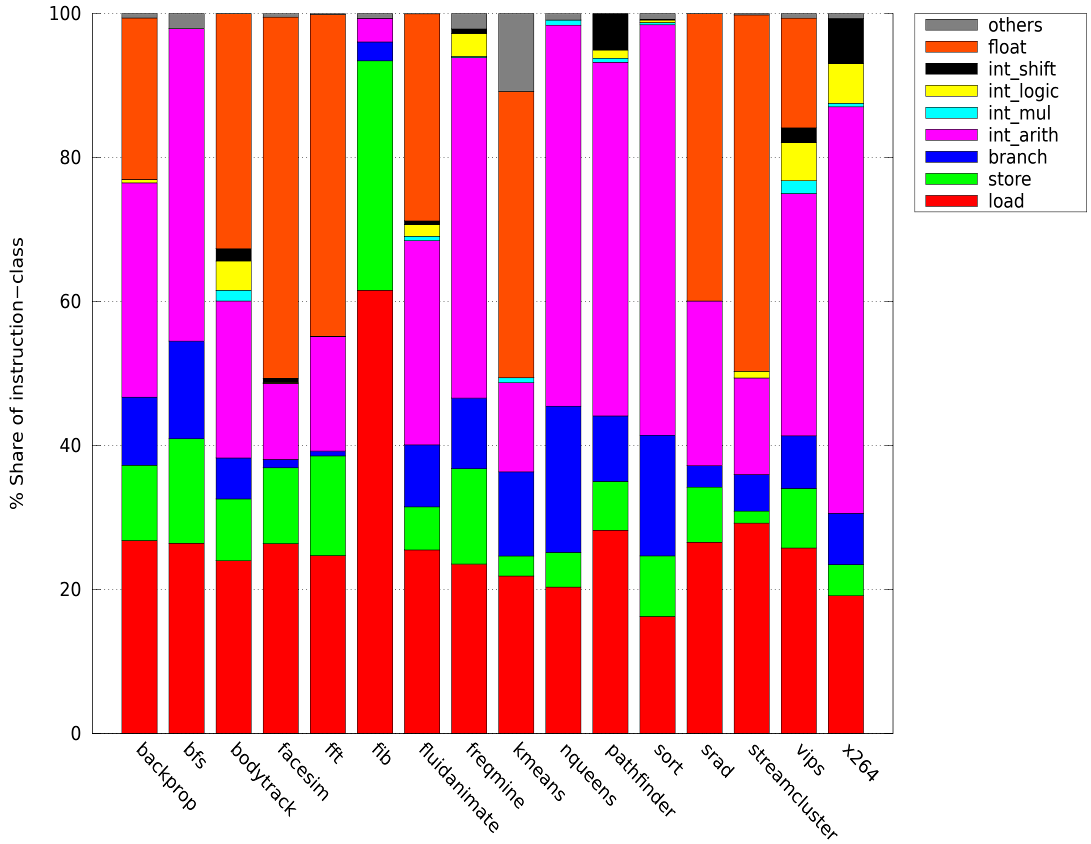


Figure 3.1: Dynamic instruction mix per application.

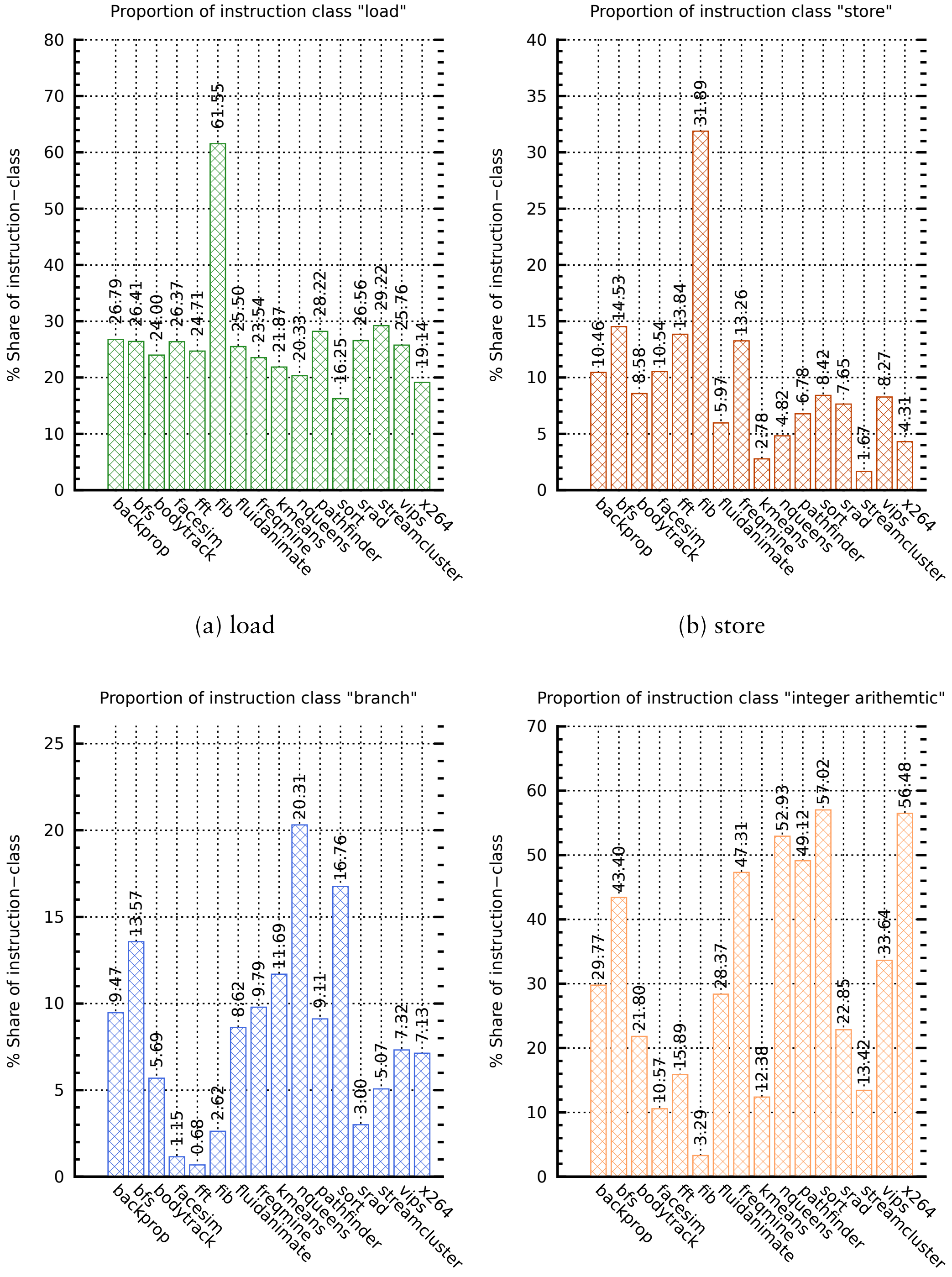
`integer multiplication`, `logic`, `shift`, `floating-point arithmetic` and `others`.

The dynamic instruction mix depends on the application, the compiler, and the processor's instruction set, but not on architectural/hardware parameters of the processor such as the number of execution units, cache sizes, etc.

Figure 3.1 depicts the distribution of the instruction classes while figures 3.2a to 3.2i, on pages 45 to 47, show dynamic instruction mix in more detail by displaying the frequencies for each instruction class individually.

3.3.1.1 Observations

The analysis of the dynamic instruction mix leads to a number of observations. First, 7 of the 16 applications do not have floating-point operations. The rest have



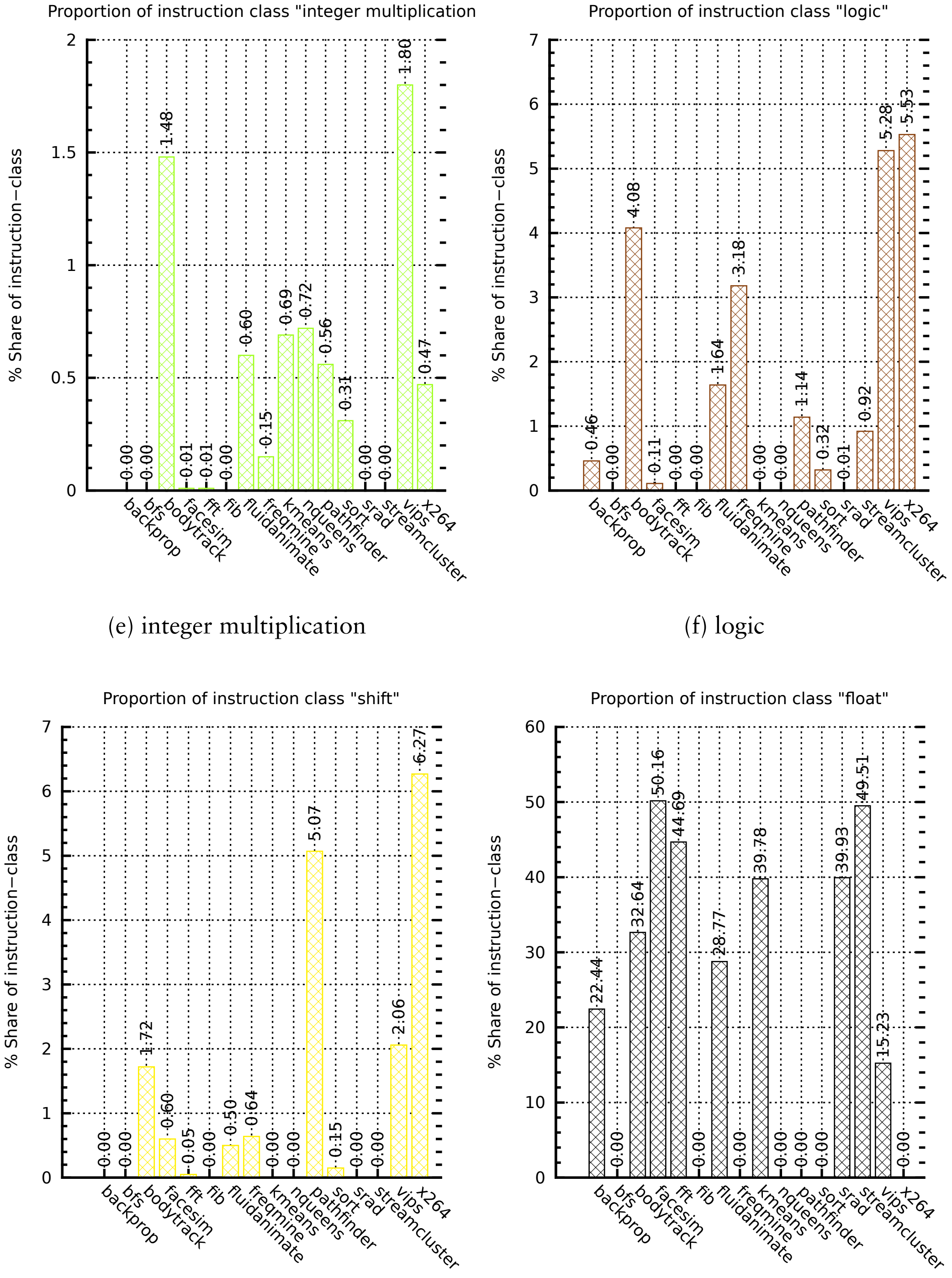
(a) load

(b) store

(c) branch

(d) integer arithmetic

Figure 3.2: Dynamic instruction mix for each individual class.



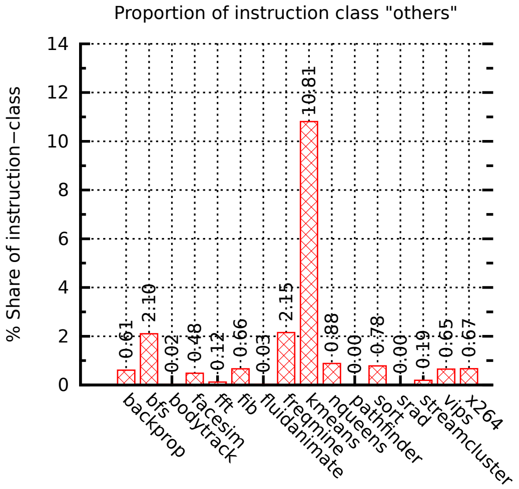
(e) integer multiplication

(f) logic

(g) shift

(h) floating-point

Figure 3.2: Dynamic instruction mix by class (cont.).



(i) others

Figure 3.2: Dynamic instruction mix by class (cont.).

a significant number of floating point operations with `facesim` having 50% of its instructions involving floating-point operands.

Another is that the individual share of `integer multiplication`, `logic` and `shift` classes are quite small. The combination of `load`, `store` and `integer arithmetic`/`floating-point arithmetic` tend to dominate in almost every application.

3.3.2 Kernel Profiling

Kernel profiling involves locating the kernels of an application that take most of the execution time. A function breakdown facilitates the identification of kernels in an application. A function breakdown lists all functions of an application together with the relative contribution of each to execution time. Figure 3.3 on page 48 is the function breakdown for the applications in table 3.1. The segments in each bar represent the dominant functions for the application except the rightmost segment which represents the merged shares of all remaining functions. For instance, in `vips` the dominant functions are `fine_gen`, `conv_gen` and `imb_XYZ2La` with 20.64%, 15.98% and 14.14% of execution time. The rest of the functions represented by the last segment are only 49.24% of execution time.

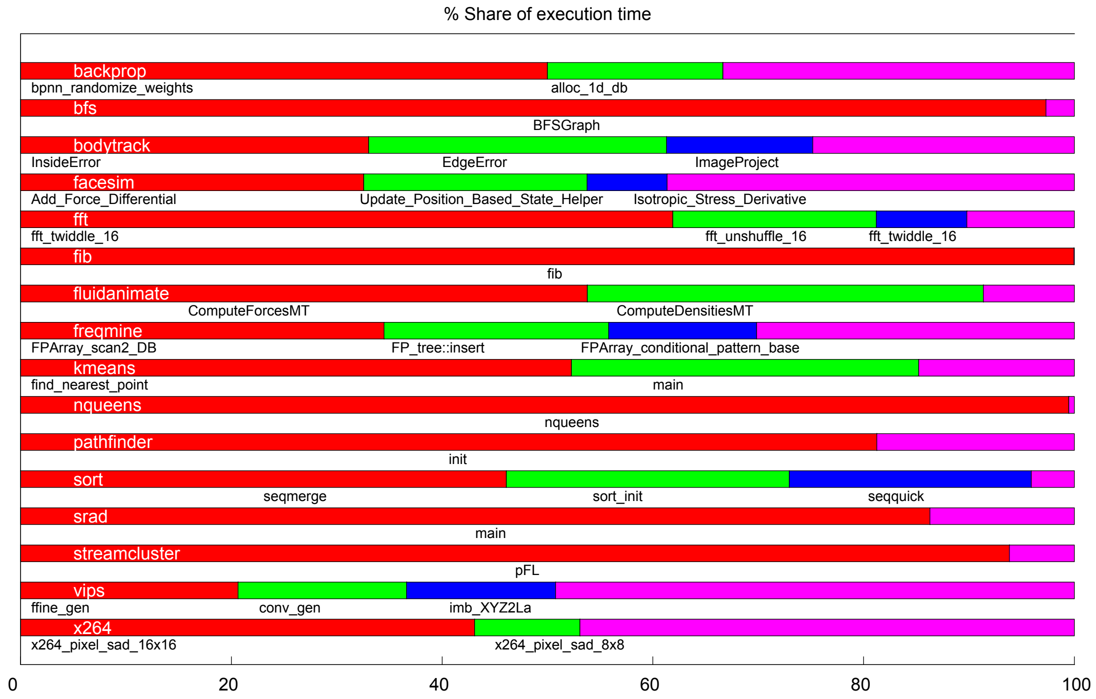


Figure 3.3: Function breakdown based on execution time.

3.3.2.1 Observations

Most applications spend a majority of their execution time inside at most two functions. There are applications, such as `fib`, where one function dominates. However, even in cases without clear dominant functions (e.g. `vips`), the two most referenced functions still account for a significant proportion of execution time.

3.3.3 Instruction Level Parallelism

ILP refers to the number of individual ‘machine’ operations executable in parallel. High ILP is important for applications targeting highly-parallel RH.

To characterize the degree of ILP within each application, an idealised out-of-order processor model is considered. Architectural features are idealised and unlimited (perfect caches, perfect branch prediction, infinite number of functional units, etc.) except for the instruction window size. ILP is measured in terms of Instructions Per Cycle (IPC). Figure 3.4, on page 50, shows the IPCs at window size of 32, 64, 128 and 256 in-flight instructions.

3.3.3.1 Observations

As expected, streaming (e.g. `x264`) and related applications (e.g. `streamcluster`)—characterised by compute-intensive numerical code—tend to have relatively high ILP. Common and important computer science algorithms such sorting (in `sort`) tend to have relatively low ILP.

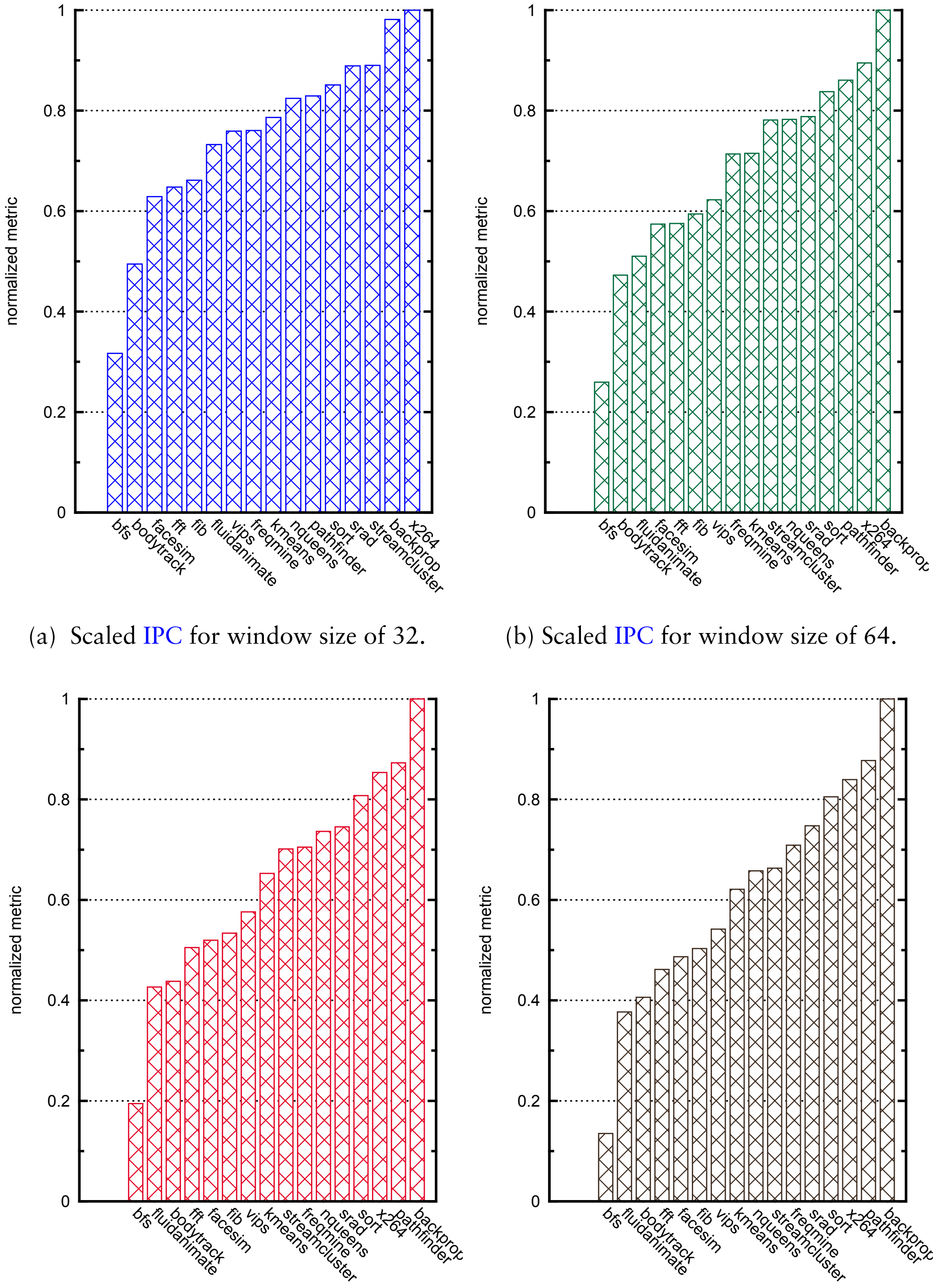
3.3.4 Branch Predictability

Branch predictability refers to how accurately future branch behaviour can be foreseen. This is important as the amount of ILP within a basic block is usually not enough to keep typical RH fully utilized. Hence, an increasing number of reconfigurable systems [24] speculate across branches. The predictability of branches determines how accurate and efficient speculation as an execution strategy will be.

The predictor used here is the Prediction by Partial Matching (PPM) predictor [84]. This allows the capture of branch predictability in a microarchitecture-independent manner as a PPM predictor can be viewed as a theoretical basis for branch prediction rather than an actual hardware predictor [82]. Four variations of the PPM predictor are considered: *GAg*, *PAg*, *GAs* and *PAs*. ‘G’ is global branch history, ‘P’ is per-address or local branch history, ‘g’ is one global predictor table for all branches and ‘s’ is separate tables per branch. The accuracy of the 4 variations of the PPM using different history lengths are given in figure 3.5 on page 51.

3.3.4.1 Observations

The branch behaviour of most applications is predictable (over 90% of the predictions are correct), with a variety of prediction schemes. However, there are a few exceptions such as `sort` with accuracy level of less than 80% across all prediction schemes used. The *GAg* scheme accuracy level tends always to lag the



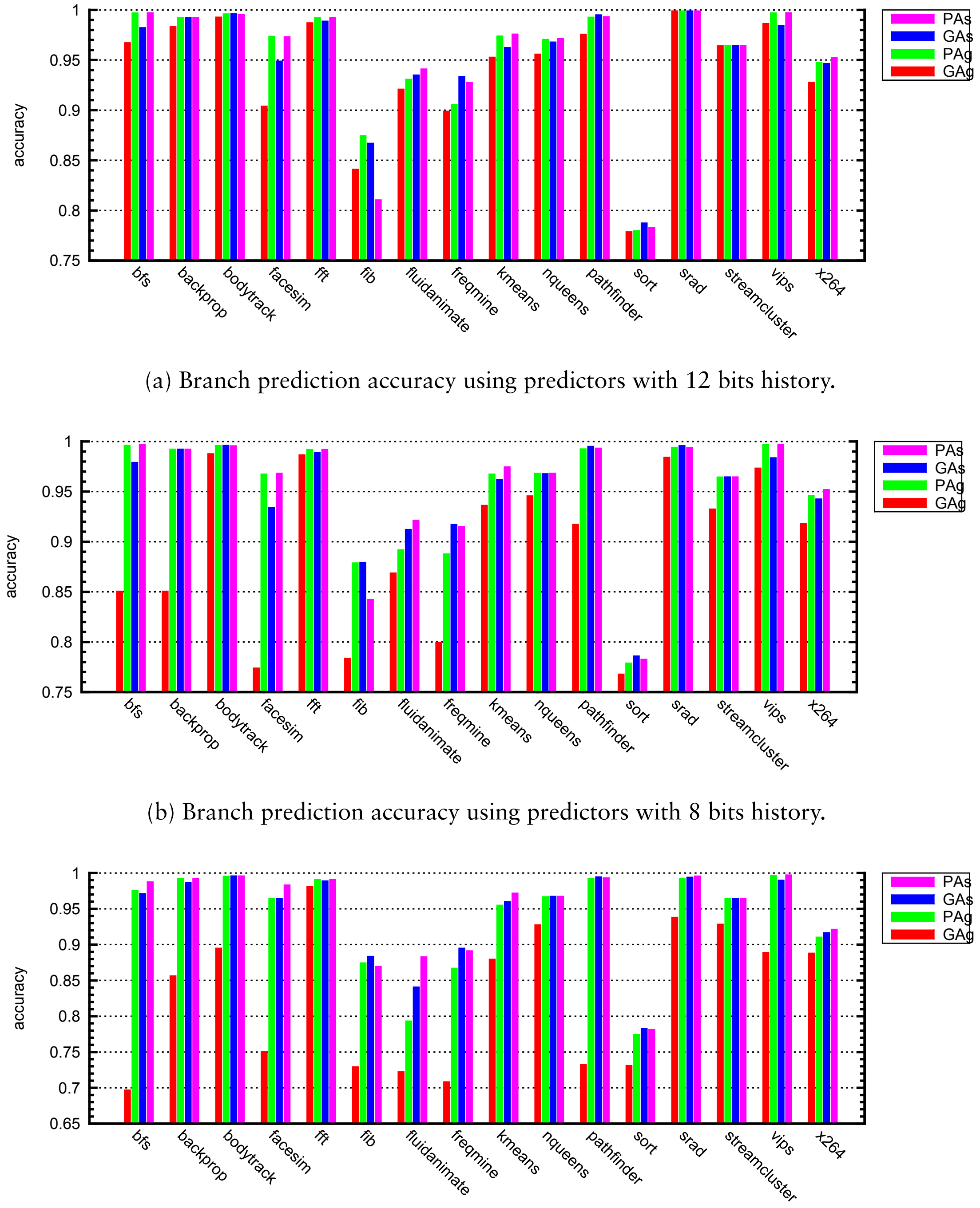
(a) Scaled IPC for window size of 32.

(b) Scaled IPC for window size of 64.

(c) Scaled IPC for window size of 128.

(d) Scaled IPC for window size of 256.

Figure 3.4: Scaled IPC for idealised processor window sizes 32, 64, 128, 256.



(a) Branch prediction accuracy using predictors with 12 bits history.

(b) Branch prediction accuracy using predictors with 8 bits history.

(c) Branch prediction accuracy using predictors with 4 bits history.

Figure 3.5: Accuracy for the 4 variations of the PPM with history lengths (12, 8, 4).

other 3 schemes and there is little difference in terms of accuracy among the other 3 schemes.

3.3.5 Memory Reuse Distances

Caching frequently used data in smaller, local buffers near the RH is essential as it reduces the access time to data [50]. The efficacy of caching depends on the principle of locality of reference. The locality of reference in any application can be quantified using the memory reuse distance. Quantifying the locality of references enables the determination of the appropriate size for the local buffers.

Memory reuse distance is the number of distinct data element references between two references to the same data element. It allows the quantitative comparison of the locality of applications that is not tied to any particular buffer design.

Address traces are obtained using MICA [82]. In MICA, reuse distances are limited to memory reads and are evaluated as follows. The address space is divided into 64-byte memory blocks. For each memory read, the corresponding 64-byte memory block is determined. Reuse distance is the number of distinct 64-byte memory blocks referenced between two references to the same block. The reuse distances for all memory reads are then sorted into 19 buckets. Each bucket captures distances within $[2^n, 2^{(n+1)}[$ 64-byte block with n ranging from 0 to 18. The first captures reuse distances of $[1, 2[$ while the last actually captures $[2^{18}, \infty[$. For example, the fourth bucket corresponds to accesses with reuse distance between 2^3 and 2^4 blocks.

Table 3.2 on page 53 shows the probability of reads to the same address having a particular reuse distance. For instance, the probability of two memory reads to the same data element in `backprop` having a reuse distance between 2^0 and 2^1 is 0.50. Note that the probabilities in the table do not add up to 1 as cold references (data elements referenced only once) are not shown in the table. Furthermore, some of the probabilities are so small that they are shown as 0.00. Columns where all rows are 0.00 are not shown in the table.

	Buckets									
Benchmark	$[0,1[$	$[1,2[$	$[2,3[$	$[3,4[$	$[4,5[$	$[5,6[$	$[6,7[$	$[7,8[$	$[11,12[$	$[18,\infty[$
backprop	0.50	0.09	0.04	0.13	0.17	0.02	0.00	0.00	0.00	0.05
bfs	0.56	0.09	0.11	0.05	0.06	0.06	0.07	0.00	0.00	0.00
bodytrack	0.66	0.11	0.11	0.05	0.03	0.01	0.01	0.00	0.02	0.00
facesim	0.68	0.07	0.08	0.07	0.03	0.02	0.02	0.01	0.00	0.01
fib	0.58	0.08	0.11	0.09	0.03	0.04	0.04	0.01	0.00	0.00
fluidanimate	0.58	0.13	0.15	0.04	0.04	0.03	0.02	0.00	0.00	0.00
fft	0.58	0.10	0.09	0.06	0.04	0.02	0.05	0.00	0.00	0.02
freqmine	0.45	0.12	0.14	0.11	0.06	0.07	0.04	0.01	0.00	0.00
kmeans	0.83	0.01	0.02	0.05	0.02	0.06	0.00	0.00	0.00	0.00
nqueens	0.34	0.63	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00
pathfinder	0.54	0.10	0.08	0.09	0.07	0.05	0.07	0.00	0.00	0.00
sort	0.76	0.15	0.02	0.01	0.00	0.00	0.00	0.00	0.00	0.03
streamluster	0.90	0.01	0.01	0.00	0.00	0.05	0.00	0.00	0.00	0.03
vips	0.44	0.23	0.18	0.06	0.04	0.04	0.00	0.00	0.00	0.00
x264	0.74	0.12	0.06	0.02	0.02	0.02	0.01	0.00	0.00	0.00

Table 3.2: Memory reuse distance statistics.

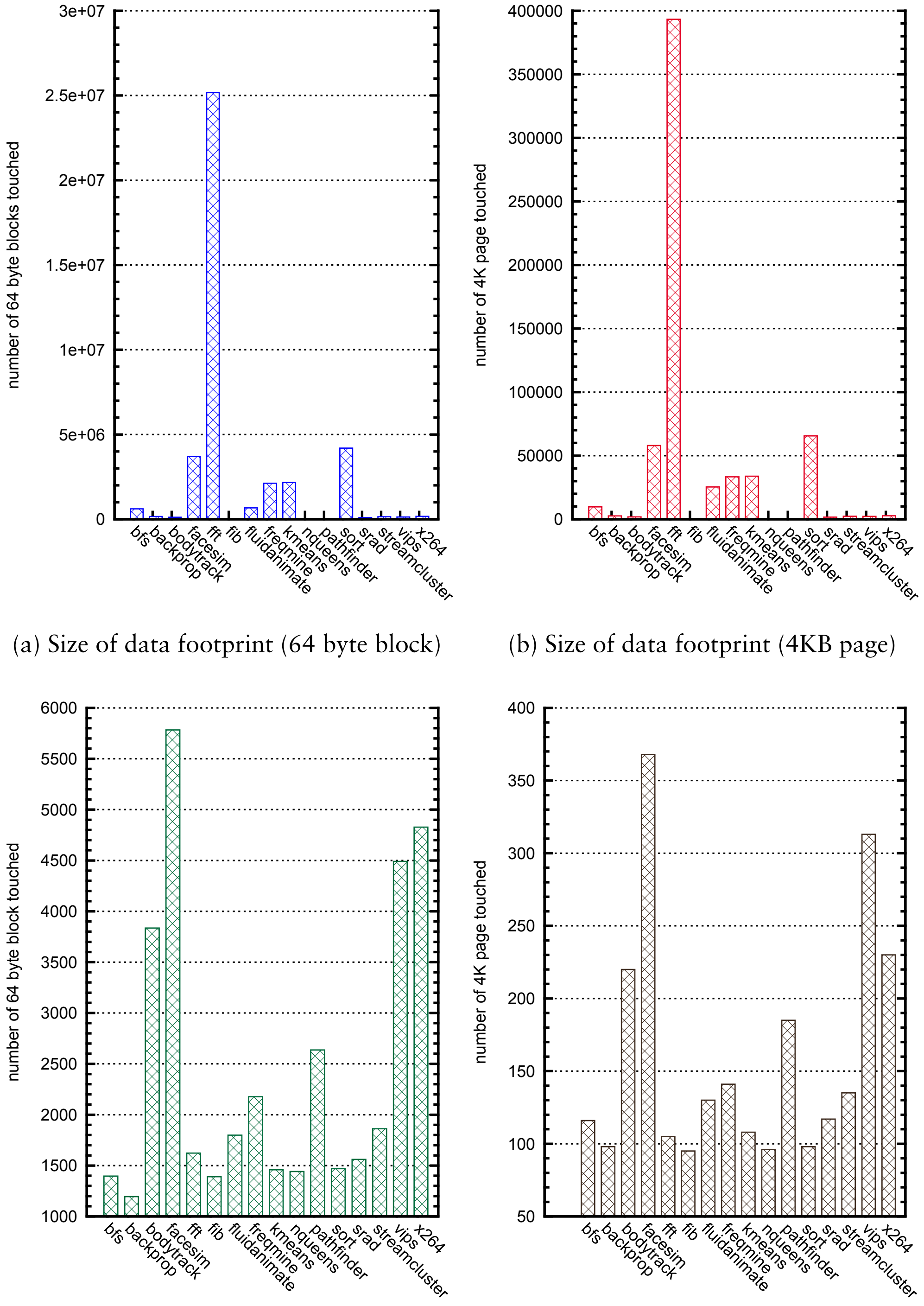
3.3.5.1 Observations

Most memory reads from the applications in the pool tend to fall within distance buckets $[0,1[$ and $[1,2[$. For a majority of applications the $[0,1[$ bucket covers more than half of all memory reads.

3.3.6 Memory footprint

Memory footprint refers to the amount of memory referenced by an application during execution. Quantifying memory footprint is important for mobile processor design as memory issues play a vital role, and often impact significantly performance, power consumption and overall cost of a design [85].

Figure 3.6 on page 54 shows the instruction and data memory footprint for each application in the application pool. Memory footprint is characterized by counting the number of unique 64-byte blocks and 4KB pages touched. The counts for data and instruction addresses are presented separately.



(a) Size of data footprint (64 byte block)

(b) Size of data footprint (4KB page)

(c) Size of instruction footprint (64 byte block)

(d) Size of instruction footprint (4KB page)

Figure 3.6: Size of Data and Instruction Memory Footprints.

3.3.6.1 Observations

fft has a relatively large data footprint (large working set) compared to the other applications. However, its instruction footprint is small as it repeatedly applies a small set of instructions over a large amount of data. Streaming (e.g x264 and related applications (e.g streamcluster) tend to have larger instruction footprints than the rest.

3.3.7 Parallel Speedup

Figure 3.7 on page 56 shows the parallel speedup for each reference application. This indicates the number of processors that can be effectively utilize by an application assuming a perfect memory system and communication architecture. Hence, the speedups in Figure 3.7 could be considered as upper bounds which may not be achieved in a real machine. This study did not attempt to account for the effects of load balancing and simply ran applications with the recommended ratio of threads to CPU. Speedups are across parallel regions of code only and are measured by counting the number of executed instructions.

3.3.7.1 Observations

Only a few of the applications have speedups close to the ideal. The difference between the ideal and average speedup becomes significant beyond 8 cores. For instance, the average speedup at 8 cores is only 60% of the ideal. The non-linear speedup shown by most of the applications suggests that serial sections are still significant.

3.3.8 Key Characteristics

- 3 classes of instructions—load, store and integer arithmetic or load, store and floating-point arithmetic—tend to dominate in every application.
- A majority of applications require floating-point support.
- A few functions dominate execution in most of the applications.

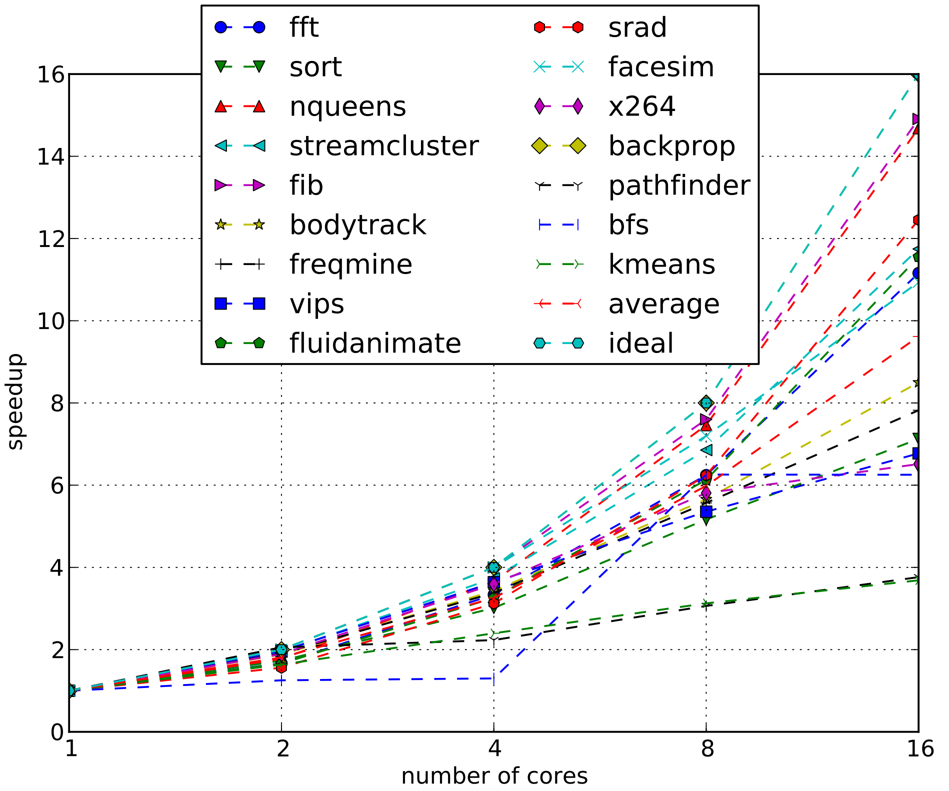


Figure 3.7: Speedup for 1, 2, 4, 8 and 16 cores.

- Branch behaviours are largely predictable.
- Memory reuse distances for reads are largely within 1 and 4 64-byte memory blocks.
- Unsurprisingly, speedup via parallelization is not linear even when restricted to parallel code regions.

3.4 Impact on Reconfigurable Accelerator Design

This section highlights how the application characterisation guides the design of VIREMENT. The actual design is discussed in chapters 4 and 5.

- Sequential sections of applications are significant as indicated in section 3.3.7. This confirms the need for efficient single-thread acceleration in mobile CMPs, as was suggested in chapter 1.

- The x86 architecture, the platform used during characterisation, has limited bit-operators. As such, applications that do extensive bit-level computation have to use a lot of `shift` and `logic` instructions to synthesise bit-operators. However, the analysis of the application pool shows that shifts and logic operations are relatively small, indicating that the majority of operations are at the word level. This supports employing a coarse-grained RH in the proposed architecture.

  The proposed RH should support the dominant instruction classes: `load`, `store`, `integer arithmetic` and `floating-point arithmetic`. However, `shift` and `logic` instructions are often intertwined with other classes of instructions. Not supporting them may reduce the number of operations that could be executed on the RH.

- The amount of ILP among the applications varies suggesting a two dimensional array of interconnected PEs (see section 3.3.3 on page 48). A 2D array exploits ILP, when and if available, but is also capable of executing sequences of dependent instructions. This, potentially, allows the same hardware to accelerate diverse applications such as `pathfinder` (with relatively high ILP) and `bfs` (with relatively low ILP).

3.5 Summary

This chapter characterised a number of reference applications with the aim of using the output of the characterisation process to guide the design of VIREMENT. The following characteristics were quantitatively measured: dynamic instruction mix, dominant kernels, ILP, branch predictability, memory reuse distances and parallel speedup. How these impact the design of VIREMENT was discussed. The next chapter discusses the design of VIREMENT.

4

VIREMENT Hardware Architecture

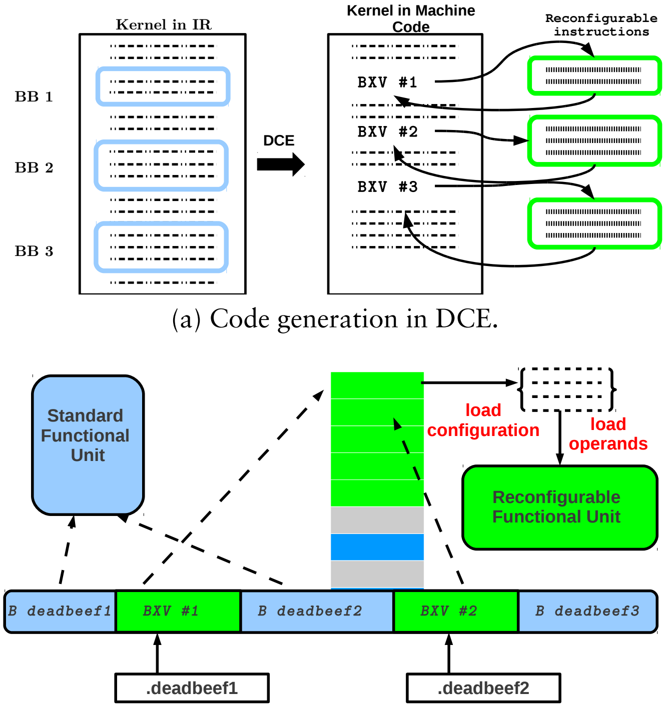
4.1 Introduction

VIREMENT could be described as a multicore dynamic ASIP [86] that uses Reconfigurable Functional Units (RFUs) instead of the custom functional units to boost single thread performance. Application sections targeting the RFU are programmed with the reconfigurable instruction set, microops, instead of the standard instruction set. The microops are hardware primitives, similar to microcode [87], for orchestrating processing on the RFU.

This chapter starts with a high level description of how the VIrtual REconfigurable Micro-ENgine for Translation (VIREMENT) and its associated JIT complier, the Dynamic Compilation Engine (DCE), improve single thread performance. It then delves into the details of VIREMENT's design. The DCE is described in chapter 5.

4.2 VIREMENT Overview

VIREMENT executes instructions like conventional processors but, additionally, adapts to the executing application at runtime with Reconfigurable Hardware (RH). The flexibility of the RH, integrated as an extra functional unit, reduces its efficiency compared to an ASIP but enables its use as a *programmable general purpose* accelerator. The DCE, based on Lower Level Virtual Machine (LLVM) [59], generates microops for programming the accelerator on-the-fly enabling



(a) Code generation in DCE.

(b) Reconfigurable execution in VIREMENT.

Figure 4.1: Overview of compilation and execution in VIREMENT.

VIREMENT to adapt to dynamically generated code.

The code generation process is quite simple and is illustrated at a high level in figure 4.1a. Initially, critical functions (kernels) requiring acceleration are identified, translated into a suitable IR and presented to the DCE by the VM that wants its code accelerated on the RH. Each basic block in a critical function is then translated into microops by the DCE. The mapped basic block is replaced in the IR by a single special instruction which serves as a pointer to the location of the microops for that particular basic block. Finally, the modified IR is complied to the target binary by the DCE. Each original basic block is now replaced by the special instruction pointing to its configuration and executes as an atomic unit on the reconfigurable functional unit as in figure 4.1b. Essentially, the DCE synthesises an application specific instruction (or complex custom instruction), on-the-fly, to replace each basic block in a kernel. Large basic blocks may, albeit, map into more than one group of microops.

Reconfigurable instructions can also be generated statically for VIREMENT but applications written in this manner are no longer portable as the reconfigurable ISA is not intended to be backward compatible. However, static compilation could be exploited to develop highly optimised software in situations where forward-scaling is not a design objective.

4.3 Microarchitecture

VIREMENT assisted by the DCE is required to increase single-thread performance in the presence of dynamically generated code by synthesising (on-the-fly) and executing complex custom instructions. Traditionally, complex instructions in Reconfigurable Processors (RPs) are built from multiple basic blocks to increase the performance impact and amortise the cost of reconfiguration. Combining basic blocks into more complex structures using traditional complier techniques, such as superblock scheduling [88] and trace scheduling [89], increases the number of operations and ILP among the operations. Building such structures at runtime, however, is expensive. Therefore, VIREMENT has to work with the small number of operations and ILP available within a basic block. As such, each complex instruction synthesised will have a small execution period. This implies frequent switching between the CPU and the RH.

The relatively short execution time of the complex instructions and the frequent switching between standard and reconfigurable execution suggests that optimal performance will be achieved by integrating the RH as another functional unit within the processor. Compared to other techniques of integration, an RFU has the lowest communication latency and is the most versatile (see chapter 2). Hence, each core in VIREMENT is a host CPU augmented with a RFU, the VIREMENT Reconfigurable Functional Unit (VRFU).

4.3.1 CPU

The CPU is comparable to ARM926EJ-S [90]. It is a simple, in-order, 5-stage pipelined, modified Harvard architecture, RISC core. It can be replicated n times to form a CMP (see figure 4.2 on page 61). The L1 caches are 32 KB and the design is discussed in page 70. Coherence is maintained among the L1 data caches

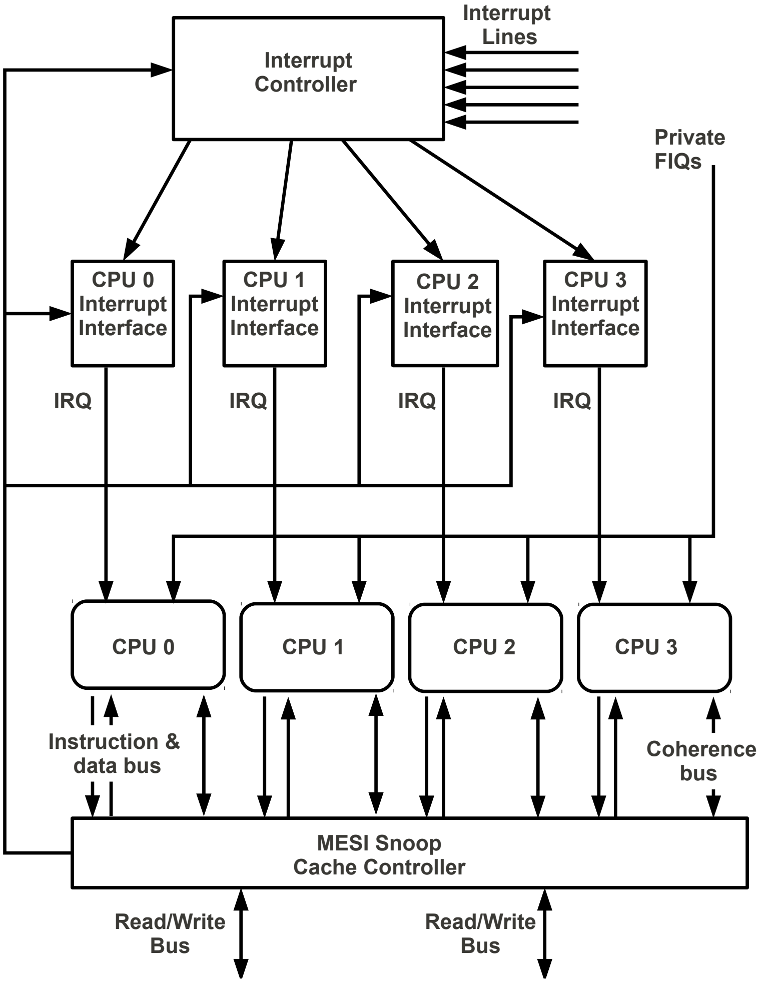


Figure 4.2: 4-core VIREMENT processor block diagram.

by an MESI [91] protocol snooping cache controller. The processor implements a weak ordering memory consistency model [92].

Each core supports three instructions sets: ARM, Thumb and VIREMENT Execution Environment (VEE). ARM is the 32-bit main instruction set while Thumb is a subset of ARM with each instruction encoded in 16 bits. VEE is the reconfigurable instruction set, microops, but can only be accessed by executing a special 'ARM' instruction "Branch-to-Virement"(BXV). The actual 'decoding' of microops is done by the VRFU. Each BXV only serves as a pointer to a single context of microops, uniquely identified by the address encoded in each BXV instruction.

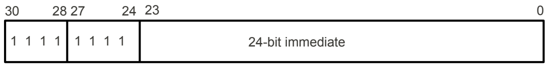


Figure 4.3: BXV binary encoding.

The BXV instruction, mapped from the ARM's system call space, depicted in figure 4.3, has a 24-bit immediate field that is forwarded to the VRFU by the ARM decoder. The VRFU interprets the immediate as a 24-bit signed offset to the memory address of the first word in a configuration (the actual mechanism is discussed in section 4.4.2.1 on page 72).

All the general purpose registers and the flag bits of the CPU are hard-wired (i.e. available) to the VRFU. The actual selection of registers is by the VRFU. Hard-wiring allows the VRFU access to all the general purpose registers in the CPU without complicating the hardware design.

4.4 Organization and Integration

The VRFU consists of a two dimensional array of processing elements (see page 56), the VIREMENT Reconfigurable Datapath (VRD) and the VIREMENT Control Unit (VCU). The VCU is mainly responsible for managing reconfiguration feeding the VRD with the correct set of microops as selected by the address encoded in a BXV.

Figure 4.4 on page 63 shows how the VRD is integrated into the CPU. The numbers indicate the logical steps involved in processing a BXV instruction and are explained below:

1. The ARM decoder partially decodes a BXV and stalls further instruction fetches. The address section of the decoded instruction is forwarded to the VCU. The CPU now awaits the completion signal from the VRFU.

2. The VCU fetches, from memory, the correct set of microops using the address from the previous step.

3. The VRD selects the particular registers needed from the hard-wired register lines.

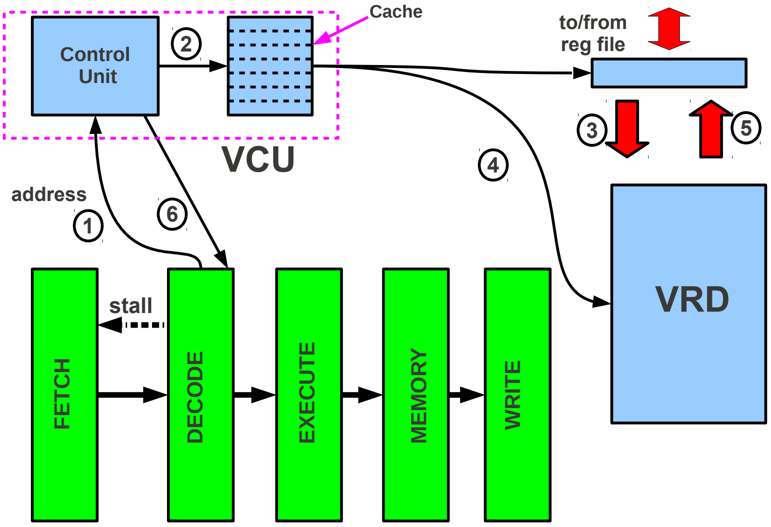


Figure 4.4: High level description of instruction execution on VRFU.

4. Execution starts on the VRD under the supervision of the control unit.

5. When execution completes the results are written back to the register file.

6. Finally, a completion signal is sent to the decoder which then removes the pipeline stall.

4.4.1 VIREMENT Reconfigurable Datapath

The VRD consists of an array of interconnected simple PEs with data routed using multiplexers (see figure 4.5 on page 65). Each PE is basically an ALU. The PEs are arranged into rows and columns, with each row connected to the next through a switch box (multiplexers). Computation flows from top to bottom with each switch box capable of connecting any of the previous row's outputs to any of the next row's inputs. This restrictive interconnection is one of the keys to enabling dynamic compilation on VIREMENT as it greatly simplifies the P&R algorithm.

The 'traditional' reconfigurable array often has, at least, mesh connection (there are connections between destinations and sources of adjacent PEs in horizontal and vertical directions). P&R on these arrays has been shown to be a NP-complete problem [93, 94] and in practice a two step process: first the placement is performed and then routing. Placement is often based on computationally intensive meta-heuristics [95] such as genetic and simulated annealing algorithms while the routing stage is usually based on pathfinder and similar algorithms which are also computationally intensive [95, 96]. Consequently, P&R on such architectures is time and resource (on some architectures the complexity of P&R approach that of FPGAs) making them unsuitable for resource-constrained mobile processors; hence, the need for a more restrictive array design which simplifies P&R.

The VRD is pure combinational logic reducing its complexity, latency and power consumption. Each PE performs only integer ALU operations, including address generation for memory operations, keeping the whole structure simple and efficient. Floating-point operations in the VRD are not yet supported.

Floating-point operations are complex and multi-cycled, each operation requires layers of configuration. With floating-point intensive applications—a significant number of reference applications in see chapter 3 are floating-point intensive—simply using multiple configurations to perform each operation will lead to the VRD running out of configuration cache capacity (configuration caches as shown on page 72 are used to reduce reconfiguration costs), causing costly fetches of additional configurations from main memory. The development of a cost-effective technique for sharing floating-point PE may be required necessitated by the large amount of resources consumed by floating-point hardware. Possible starting points for extending the VRD for floating-point operations are discussed in chapter 8.

PEs operate on the Multiple Instruction Multiple Data (MIMD) model i.e. each is configured separately via a private context register. The content of the context register determines the operation performed by the PE. Many RH are specialised for Single Instruction Multiple Data (SIMD)-style computations. SIMD is more space efficient than MIMD as a single configuration is shared by multiple PEs. This style is well suited for streaming type programs, where data parallelism abounds, saving configuration and cache storage by sharing an instruction for multiple data. In the absence of abundant data parallelism performance suffers on SIMD arrays as each PE cannot execute independently. VIREMENT largely maps a small number of independent operations at a time to the VRD making

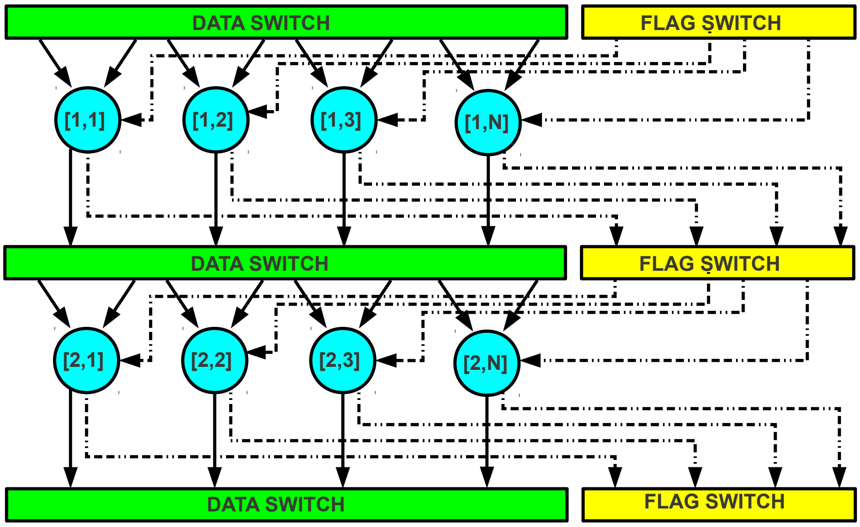


Figure 4.5: ALU interconnection on the VRD.

MIMD-style more appropriate.

4.4.1.1 ALU

The fundamental processing element in the VRD is the ALU. The VRD's ALUs are kept simple to reduce latency, cost and energy consumption. Each ALU operation can set 4 1-bit flags: a *N-sign flag* (set if operation result is negative), a *Z-zero flag* (set if adding two operands of the same sign yields a result with opposite sign), an *V-overflow flag* (set if number exceeds range) and a *C-carry out flag* (set if there is a carry out of the most significant adder result bit). The flag structure is similar to the one on the host CPU allowing the CPU and VRFU to exchange flags.

Each operation has three operands, two being 32-bit values and the third a 1-bit value. The third operand can be a flag from any of the ALUs in a previous row. Each of the two 32-bit values can be from any of the ALUs in the previous row, any of the host processor's general purpose registers or an 8-bit immediate value encoded in the microop itself. other instruction in the ISA.

Supported ALU operations are listed in table 4.1 on page 66. The operations selected were determined with the help of the dynamic instruction mix of reference applications (see page 56).

Opcode	Description
add	$op1 + op2$
addc	$op1 + op2 + flag_C$
sub	$op1 + \overline{op2} + 1$
subc	$op1 + \overline{op2} + \overline{flag_C}$
lshl	$op1 \ll_{logical} op2$
lshr	$op1 \gg_{logical} op2$
ashl	$op1 \ll_{arith} op2$
ashr	$op1 \gg_{arith} op2$
and	$op1 \wedge op2$
or	$op1 \vee op2$
xor	$op1 \veebar op2$

Table 4.1: Supported data operations.

4.4.1.2 Interconnect

The interconnect links the ALUs together to obtain the desired functionality. There are no links between PEs on the same row to reduce the complexity of P&R and the hardware. Figure 4.6 on page 67 shows how the data switches (from figure 4.5 on page 65) are composed from multiplexers. Flag switches, omitted for clarity, are similar to the data switches. Since register inputs are hard-wired, see section 4.3.1 on page 60, multiplexers select the registers for each ALU. Another multiplexer, connected to the output of each ALU, selects the register line on which the ALU result will continue.

In some situations, e.g. multi-stage operations using temporary values, the result from an ALU operation should not be written to the register lines but forwarded to another ALU. To handle such situations, the second operand of each ALU can be from any ALU in the previous row. Figure 4.7 on page page 68 shows the ALU with this enhancement. A similar arrangement is used to forward flags.

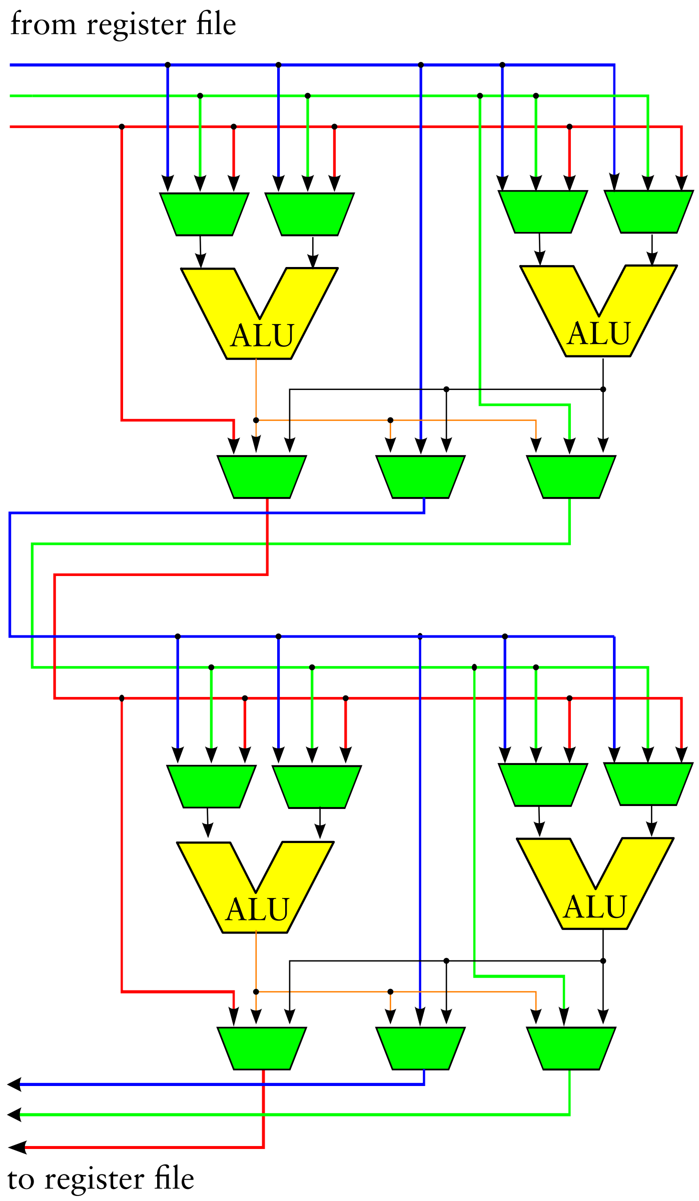


Figure 4.6: Illustrating the interconnect network using a $2X2$ VRD.

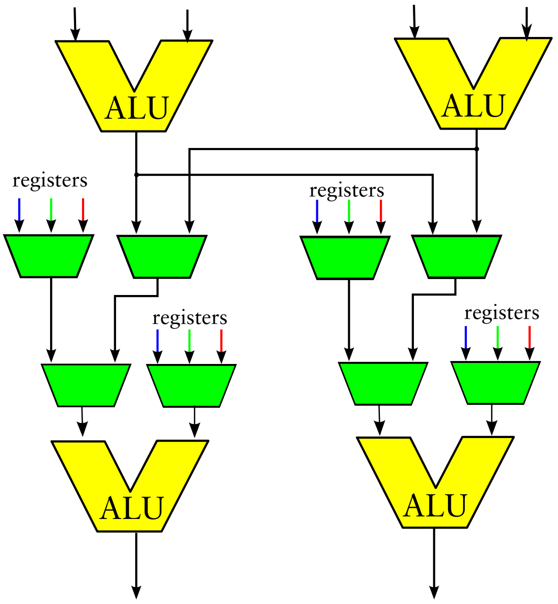


Figure 4.7: Cut away of VRD showing result forwarding multiplexers.

4.4.1.3 VRD Dimensions

To determine the number of columns (width) and rows (depth) for the VRD LLVM Out-of-Order-Processor Emulator (LLVMOOPE), developed as part of this thesis, is used. LLVMOOPE is a custom LLVM tool, based on *lli* (LLVM byte code interpreter), that mimics an Out-of-Order-Processor (OOP) processor model in which everything is idealized and unlimited except for the window size (limited to a basic block). It takes basic blocks from programs in LLVM bitcode, converts LLVM instructions into microops, schedules these microops on the ideal OOP and outputs the schedule in a row/column format termed a schedule matrix (see figure 4.8). A schedule matrix is basically a trace of instructions executed in the ideal processor presented as an x-by-y matrix. The x dimension (width) is the number of instructions executed in parallel while the y dimension (depth) represents cycles (time). The functional units of the out-of-order processor supports only the operations listed in table 4.1 on page 66. Any LLVM instruction that can not be fused from (or mapped to) the microops in table 4.1 triggers the creation of a new schedule matrix.

A schedule matrix approximates the dimensions of a VRD needed to execute

```
c = (a+8)+(a-4)+ b;
b = b+3;
```

(a) LLVM code.

`i1:`	`t1 = ldr a`
`i2:`	`t2 = ldr b`
`i3:`	`t3 = t1 + 8`
`i4:`	`t4 = t1 - 4`
`i5:`	`t5 = t2 + 3`
`i6:`	`t6 = t4 + t2`
`i7:`	`t7 = t3 + t6`
`i8:`	`c = str t7`
`i9:`	`b = str t5`

(b) Microops Code.

cycle	FU 1	FU 2	FU 3
0	`t1 = ldr a`	`t2 = ldr b`	
1	`t3 = t1 + 8`	`t4 = t1 - 4`	`t5 = t2 + 3`
2	`t6 = t4 + t2`	`t7 = t3 + t6`	`b = str t5`
3	`c = str t7`		

(c) Schedule Matrix.

Figure 4.8: Basic block processing in LLVMOOPE.

the basic block from which the matrix was generated. LLVMOOPE compared to the VRD, makes a number of simplifying assumptions, such as not modelling the interconnect by assuming that routing from a producer to a consumer is guaranteed by storing intermediate values in a central register file. As such, the sizes of schedule matrices should be viewed as theoretical limits which may not be achieved on the VRD.

4.4.1.4 Depth

Table 4.2 on page 70 shows the cumulative percentage of schedule matrix depths for a number of the integer applications from the application pool (see page 40). For example, the 86.20% in `bfs` at depth 2 means that 86.20% of the schedule matrices in `bfs` have depths less than or equal to 2.

4.4.1.5 Width

Table 4.3 on page 70 shows the schedule matrix width statistics for 5 integer applications from page 40. For instance, only 0.81% of schedule matrices have

Depth	bfs	fib	freqmine	nqueens	pathfinder	sort	Average
1	65.2	0	46.69	62.5	78.05	60.61	47.06
2	86.21	100	78.86	81.25	87.80	84.85	86.25
3	93.10	100	92.51	87.5	97.57	90.91	92.80
4	100	100	98.56	100	97.57	90.91	97.89
5	100	100	99.71	100	100	96.97	98.34
≥ 6	100	100	100	100	100	100	100

Table 4.2: Cumulative percentage of schedule matrices with varying depths.

	1	2	3	4	5	6	7	8	9	10	11	12
1	100.00	72.40	53.40	37.02	25.96	16.38	9.79	5.53	4.04	2.55	1.70	1.49
2	47.97	22.19	12.26	7.66	2.96	1.48	1.48	1.48	1.63	0.86	0.00	0.00
3	19.27	6.90	2.23	1.42	0.81	0.00	0.00	0.00	0.00	0.00	0.00	0.00
4	7.28	1.07	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
5	1.92	0.21	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
6	0.43	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 4.3: Matrix utilization of schedule matrices.

width 5 or more in row 3. The intensity of the colour in each cell indicates the level of utilization (higher utilized cells have a darker colour).

4.4.1.6 VRD Size

The ideal would be to have a VRD that covers all the utilized cells in Table 4.3. However the cost of such a design would be prohibitive. Consequently, VIREMENT uses a 4×4 VRD as a compromise between performance and cost. From Table 4.3, a majority of the configurations would fit on a 4×4 VRD.

4.4.1.7 Data Memory Interface

From the analysis of applications (see figure 3.2a on page 45) memory accesses (i.e. load and store operations) are significant. Therefore, supporting memory accesses on the VRD would boost performance.

The CPU's L1 data cache is a natural communication medium between the VRFU and the main memory. Accessing memory through the L1 data cache keeps programming simple as it eliminates the need for the VRFU to manage coherency in a CMP design. Ideally, each PE should be able to perform two reads and a write to the local memory. Then, an L1 cache with 8 read ports and 4 write ports is required to service a row in the 4×4 VRD. Since the VRD is much faster than the CPU the cache may need to service requests from all the PEs concurrently for the cache not to become a bottleneck. This will require 32 read ports and 16 write ports. Caches, however, with such number of ports are impractical because of hardware, energy and delay (access time) costs. hardware arbiter between the memory requests generated by the PEs and the banks of the memory can be used to allow each PE access data in any bank. However, a multi-bank memory will not be able to provide data when there is a bank conflict, two different PEs requesting access to the same bank simultaneously. Software/complier techniques have been developed to reduce bank conflicts [97]

A more cost-effective solution employed in VIREMENT is to restrict memory to one PE per row with the PE only allowed one memory operand. To increase the bandwidth of the L1 data cache, cache accesses are time division multiplexed (virtual multi-porting). A virtual multi-port cache [98] creates the illusion of having multiple ports by clocking the port at a faster rate that the CPU and (time division) multiplexing accesses to the caches physical ports. However, virtual multi-porting may not scale as the frequency of the processor increases. Processors operating at very high frequencies cannot afford to pump (operate at a higher frequency) the data cache.

Multi-bank caches may be a better approach to multi-porting as mobile processor frequencies enter the gigahertz range. Multi-banking [98] divides the cache into multiple small banks, each single ported, allowing it to service multiple requests concurrently if they are to different banks.

Cache misses imply that the data operations with memory operands have variable delay. The exact mechanism for managing this is discussed in section 4.4.2.

4.4.2 VIREMENT Control Unit

The VCU's primary responsibility is runtime reconfiguration management. It fetches configuration bits from main memory, caches them locally, and loads them on demand into context registers. It is also responsible for coordinating execution between the VRFU and the CPU.

The main components of the VCU are the Execution Engine (EE) and the Reconfiguration Cache (RC). The EE is a dedicated hardware sequencer responsible for managing execution on the VRFU. The RC is an on-chip SRAM memory (and associated control logic) for caching configuration contexts. The RC allows the almost instantaneous loading of configurations into context registers.

4.4.2.1 Reconfiguration Cache

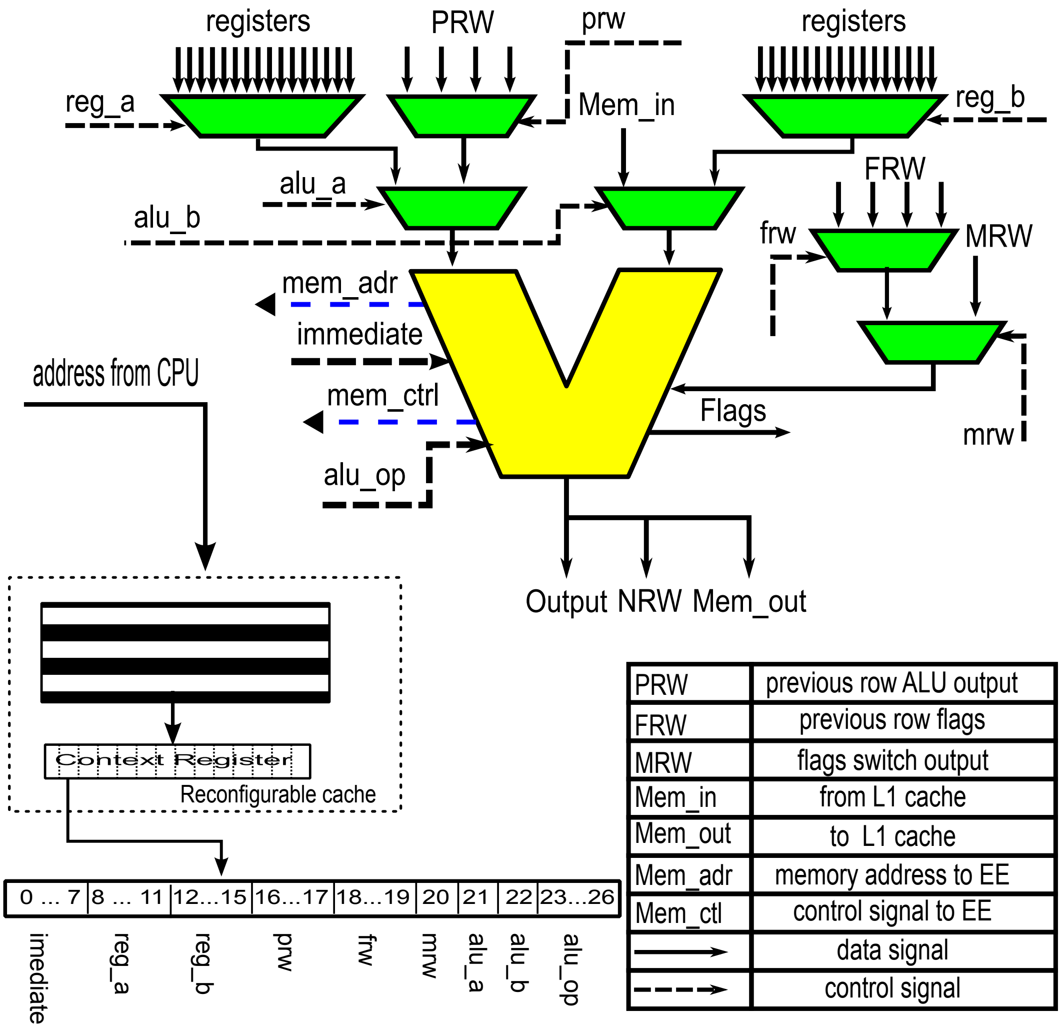
The RC is essentially a fully associative cache. Each cache slot is indexed by the address of the first memory word in a configuration. The address of the first word in a context is calculated by: (1) sign extending the 24-bit signed offset specified in BXV to 32 bits, (2) shifting the result two bits to the left to form a word offset and (3) adding it to the contents of the PC. This is the same mechanism used to determine the target of other branch instructions.

A cache miss is serviced by the main memory via DMA (set to burst mode). The DMA controller is given the start address of the microops and the number of bytes to fetch. Contexts are padded to align them to a memory word boundary and are of the same size regardless of the actual number of PEs used, keeping the RC design simple at the cost of memory space. The cache employs a LRU replacement policy.

Figure 4.9 on page 73 depicts how control signals are generated for a PE. Each PE requires a minimum of 20 control bits (figure 4.10 on page 73 shows the exact number of control bits per PE) while the data and the flag switches that move data from one row to another (see figure 4.6 and figure 4.5) require 51 bits per stage. 3 control bits per 5:1 multiplexer for each of the CPU's 16 general purpose registers and the status register. The entire 4×4 VRD requires 393 control bits.

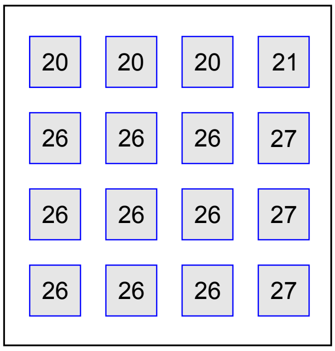
4.4.2.2 Execution Engine

The EE, depicted in figure figure 4.11 on page 74, manages the VRFU. It determines when to signal the CPU that execution on the reconfigurable array is valid. Basically, the EE loads a configuration into the context registers, waits for results on the register lines to become valid and then signals the CPU.



PRW	previous row ALU output
FRW	previous row flags
MRW	flags switch output
Mem_in	from L1 cache
Mem_out	to L1 cache
Mem_adr	memory address to EE
Mem_ctl	control signal to EE
⟶	data signal
- - -→	control signal

Figure 4.9: PE with all control bits and signals.



20	20	20	21
26	26	26	27
26	26	26	27
26	26	26	27

Figure 4.10: Control bits per PE on the 4×4 VRD.

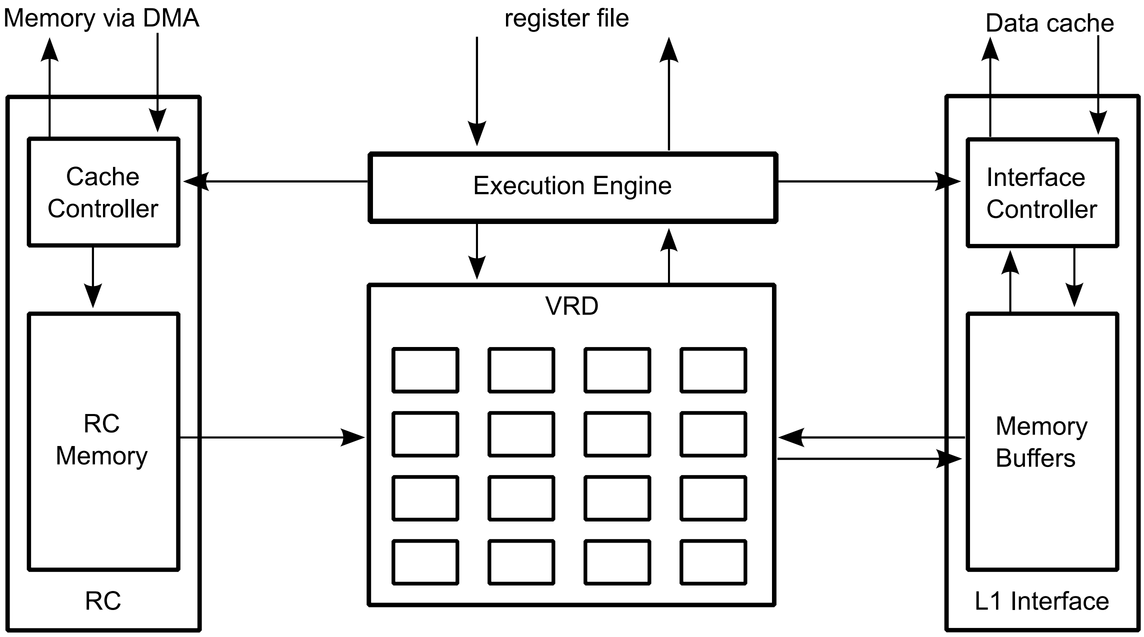


Figure 4.11: Block diagram of the VRFU.

The EE uses a counter to determine when to signal the CPU that data on the register lines are valid. Once a configuration is loaded, the EE sets the counter to a value such that when the counter zeros the results on the register lines are valid.

The EE orchestrates memory accesses between the VRD and CPU data cache. The EE has 1 store buffer per row of the VRD, which is unified into a single 4-entry array. All stores reside in the buffer until written back to data cache. This allows state to be restored easily if the block terminates unexpectedly. The EE services a load request from the store buffer, if the requested address exits in the store buffer. Since the cache is physically single-ported, the EE buffers data from it on behalf of each of the PEs that use memory operands. On a cache miss, the EE simply restarts the execution by loading a value into the counter used in timing when VRD data becomes valid. The value depends on the row of the PE that triggered the cache miss.

4.5 Summary

This chapter presented a system overview of VIREMENT followed by a detailed discussion of its design. The next chapter discusses the design of the DCE, the JIT compiler that maps applications onto the VRD.

5 Dynamic Compilation Engine

5.1 Overview

The Dynamic Compilation Engine (DCE) is VIREMENT's dynamic compiler. The necessity of dynamic compilation on mobile processors was explored in chapters 1 and 2. The DCE generates code on the fly for the VRFU, starting from LLVM IR [59] as shown in figure 5.1. As with time- and resource-limited run-time compilers used on battery powered mobile processors, the emphasis is on speed, small memory footprint and energy efficiency rather than code quality. The DCE relies heavily on the LLVM compiler framework for transformations and analysis.

Every dynamic compiler, even the leanest, has substantial overhead that may degrade overall performance. However, DCE targets VMs such as ArBB where kernels, which are typically few as shown in chapter 3, are the only dynamically

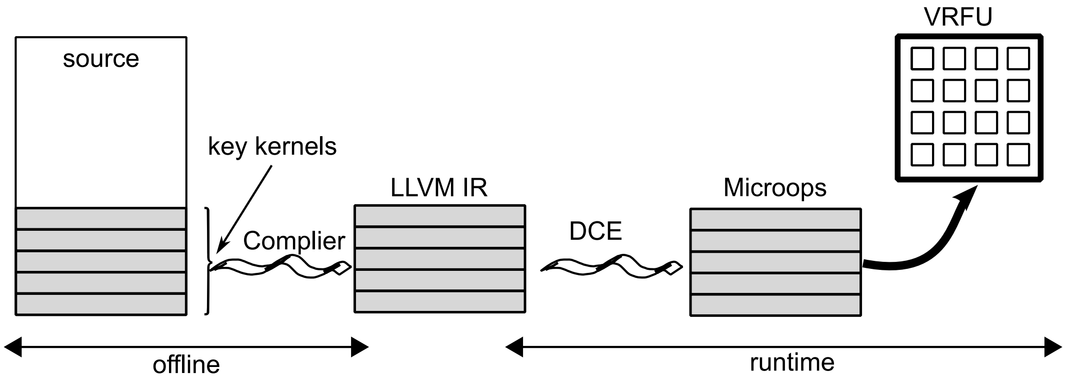


Figure 5.1: Compilation in DCE.

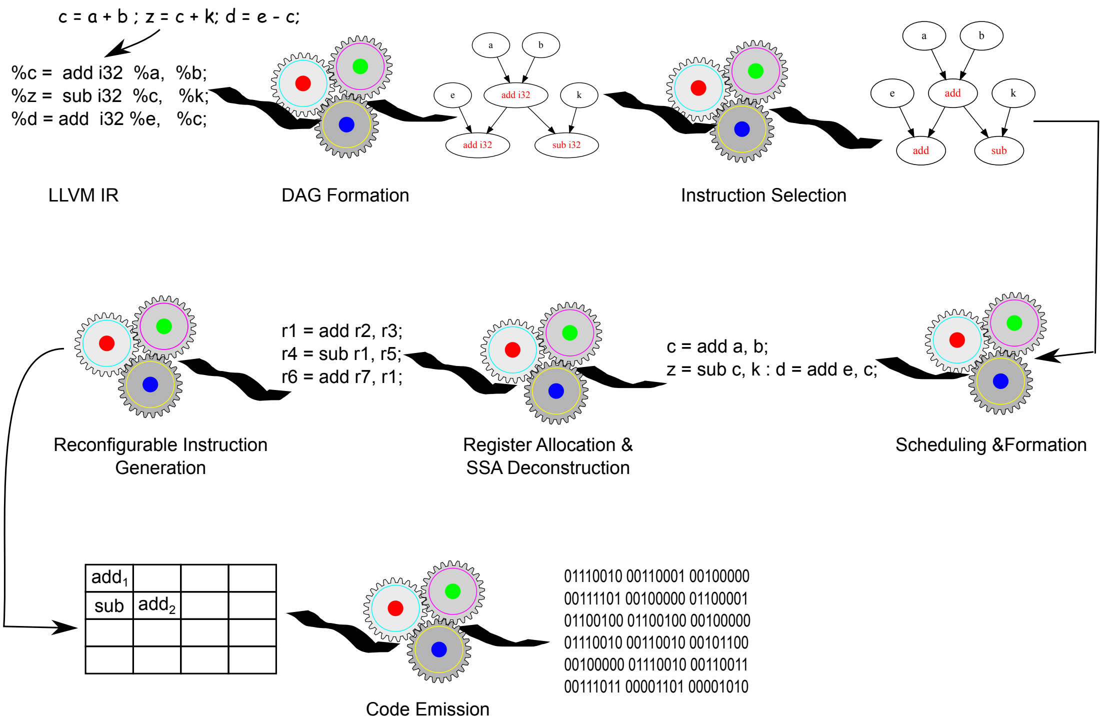
compiled sections of an application. As such, compilation overheads are largely amortised in the typical DCE usage model as only a relatively small part of the application, which runs for a relatively long period of time, is complied. Quality issues can be tackled with split compilation [99], performing time consuming analyses offline and saving the results for runtime use. However, this has not been implemented in DCE yet and is part of the potential extensions (see chapter 8).

LLVM offers a number of advantages to the DCE. The LLVM IR describes applications with low-level, RISC-like, instructions while still retaining key high-level information for effective analysis such as explicit dataflow representation using an infinite, typed register set in Static Single Assignment (SSA) [100] form. This reduces translation overheads near to that of binary translators while still allowing for more sophisticated analyses and transformations than possible in binary translators. The IR can serve as a persistent, offline code representation and as a compiler internal representation, with no semantic conversions needed between the two. This increases the flexibility of DCE as kernels can be complied to LLVM IR offline. Finally, LLVM IR is increasingly being used in parallel compilation systems targeted by DCE making integrating DCE in such systems straightforward. For instance, AMD embeds LLVM IR source for kernels in its OpenCL Binary Image Format (BIF) 2.0 [101].

5.2 DCE's Structure

DCE leverages the LLVM target-independent code generator [102]—a suite of reusable components that can be used for translating LLVM IR to binary machine code format—for code generation.

The generation of reconfigurable instructions is a post-pass optimisation within LLVM's target-independent code generator. CPU instructions are first generated and then translated into microops. This allows for the seamless intermixing of standard and reconfigurable instructions since not all operations can be performed on the VRFU. The novel pass, developed as part of this thesis, is designed to be fast and lean (discussed later) allowing its use in mobile devices with constrained processing power and storage.



```
%c =  add i32  %a,  %b;
%z =  sub i32  %c,   %k;
%d = add  i32 %e,   %c;
```

```
c = add a, b;
z = sub c, k : d = add e, c;
```

```
r1 = add r2, r3;
r4 = sub r1, r5;
r6 = add r7, r1;
```

add_1			
sub	add_2		

```
01110010 00110001 00100000
00111101 00100000 01100001
01100100 01100100 00100000
01110010 00110010 00101100
00100000 01110010 00110011
00111011 00001101 00001010
```

Figure 5.2: A pictogram of DCE's compiler stages.

5.2.1 Runtime Code Generation Process

Runtime code generation can be logically divided into nine distinct steps (see figure 5.2):

1. **DAG Formation:** The first step is the expansion of the LLVM input into a Directed Acyclic Graph (DAG) of LLVM instructions.

2. **Instruction Selection:** This step converts the DAG of LLVM instructions into a DAG of native CPU instructions using a pattern-matching instruction selector.

3. **Scheduling and Formation:** In this step, a scheduler assigns linear order to the DAG from the previous stage. The DAG is now converted to a sequential list of native CPU instructions. These instructions are represented using *MachineInstr* [102]. *MachineInstr* is an abstract way of representing machine instructions where each instruction is simply an op-code number and

a set of operands. The list is still in Static Single Assignment (SSA) form as registers are still virtual.

4. **Register Allocation & SSA Deconstruction:** Virtual registers are eliminated from instructions and replaced with physical registers. The register allocator is a fast, lean, local allocator from LLVM that attempts to keep values in registers and reuses registers as appropriate [102]. Register allocation is accompanied by the complete deconstruction of the SSA form.

5. **Reconfigurable Instruction Generation:** This step extracts and translates suitable CPU instructions into microops, emits the microops into binary form and replaces each set of translated CPU instructions with a single BXV instruction that points to the location of the microops for that particular set of instructions. The pass is a functional-level pass i.e. it executes on each function in the program independent of all of the other functions.

   Logically it has four stages (each stage is discussed in greater detail later):

   (a) **Instruction Translation:** This stage identifies and translates suitable CPU instructions to microops. Instructions are extracted, sequentially, from the list of CPU instructions and translated into microops represented using VIREMENT Intermediate Representation (VIR). The VIR is an n-tuple consisting of an operator and operands. Each microop is given a unique number, an *ID*, as it is translated. *ID*s help in tracking dependencies between microops.

   Translation starts from the beginning of a basic block and ends at an unsupported CPU instruction or the end of a basic block. When translation stops, and the number of already translated CPU instructions is above a certain threshold, compilation proceeds to next stage, else the block is deemed not worthwhile, the translated microops are discarded and translation restarts at the next instruction (or the next basic block) beyond the unsupported one.

   (b) **Microops optimisation:** A number of optimisations could be applied to the microops at this stage. Presently, the main one is the removal of copy (register-to-register move) instructions. Copy instructions are redundant as data can be moved directly from producers to consumers.

(c) **Microops Placement and Routing:** This stage involves the simultaneous placement and routing of microops on the VRFU. The output of this stage is pseudo-assembly code for the VRFU. P&R of the microops uses a simple, single-pass greedy algorithm to keep resource consumption and overhead to a minimum. The algorithm, described later, simply takes a microop and determines, based on data dependencies and resource availability, where to place it on the VRD.

(d) **Microops Code Emission:** Binary code is emitted for the VRFU. This actually happens during code emission for the target CPU.

6. **Code Emission:** Machine code is emitted into memory ready for execution. Each BXV in the machine code points to a corresponding VRFU configuration.

5.3 Reconfigurable Instruction Generation

5.3.1 Translation

Algorithm 5.1 on page 81 is the Reconfiguration Instruction Generation (RIG) pass algorithm for translating a basic block of CPU instructions into microops. Reconfigurable instruction generation starts off with the allocation of memory space for translated microops (line 2 of algorithm 5.1). Then, each CPU instruction, represented using *MachineInstrs*, is translated into microops (lines 3 to 6 of algorithm 5.1). The translation process stops at recognizable basic block boundaries like conditional branches, indirect branches and return instructions. Function calls and unsupported instructions are also considered as basic block terminators.

The basic block by basic block mapping in the RIG pass limits it to the small number of operations and ILP available in each block. The alternative, combing multiple basic blocks, with techniques like trace [89], superblock [88], and software pipelining [103], to increase the number of operations and ILP available is

Algorithm 5.1: Algorithm for Reconfigurable Instruction Generation pass.

```
Input: MBB ;                          /* Basic-block of CPU instructions */
Output: MBB$_{BXV}$ ;                 /* Basic-block post translation */
Output: Configs$_{1,2,...,n}$ ;       /* VRFU configurations */

while CPU instructions in MBB do
    initialise_translation_buffer ;
    while CPU instruction is supported do
        translate_to_microop;
        save_microop_in_translation_buffer;
    end
    if number_translated_CPU_instrs $<$ threshold then continue;
    optimize_microops;
    place_&_route;
    if P&R fails or P&R not beneficial then continue;
    replace_successfully_routed_CPU_instrs_with_bxv;
    emit_microops_to_memory;
end
```

expensive to perform online.

An example translation of a basic block is shown in listings 5.1 and 5.2 on page 81. The PC relative branch in line 5 of listing 5.1 ends the translation. The subscript numbers in listing 5.2 are the *ID*s. *%f, %i, %r, %t* denote flag, register, immediate and temporary operands. $\%f_1$ means that a flag is supplied by instruction with *ID* 1 which means that line 1 of listing 5.2 must be placed at least one row before line 2. *%i8* is an immediate of size 8 bits.

```
1  bb12:
2  %r5 = adds %r4,%r3
3  %r3 = adc %r2,%r5
4  %r4 = ldr [%r3,-%r0]
5  br %i8
```

Listing 5.1: Translation: CPU Instrs.

```
1  %r5 = add$_1$ %r4,%r3
2  %r3 = adc$_2$ %r2,%r5,%f$_1$
3  %t1 = sub$_3$ %r3,%r0
4  %r4 = ldr$_4$ [%t1]
```

Listing 5.2: Translation: Microops.

5.3.2 Optimisation

The next step in the RIG pass is to optimise the microops, line 8 of algorithm 5.1 on page 81. However, this only happens if the number of translated CPU instructions exceed a threshold. In other words, the algorithm simply abandons further processing and goes back to line 1 of algorithm 5.1 if the number of translated CPU instructions is below the profitability threshold. The threshold depends on the particular implementation of VIREMENT targeted by the DCE.

Optimisation involves removing copy instructions i.e. register-to-register move instructions, which are redundant on the VRD. The optimiser simply walks the microops, bottom-up, removing each copy instructions and then updating the operand(s) of each microop that depends on the eliminated instruction.

5.3.3 Placement and Routing

The next step in RIG is the Placement and Routing (P&R) of the translated microops (line 9 in algorithm 5.1 on page 81). P&R maps each microop to a PE and selects the appropriate inputs for that PE. The P&R algorithm is quite simple: the first step is to retrieve the next unscheduled microop. The operands (including flags) are read to verify data dependencies. Data dependencies are tracked using a small data structure called the *Dependency table* which shows the row and column on the VRD where each operand was last defined. The columns are numbered from left to right while the rows are numbered from top to bottom. The row numbers of all the microop's source operands are compared and the operand with the highest row number determines where the microop is to be placed.

The next step is to search for a free PE on the VRD to place the microop. Resource usage is modelled with a matrix-like data structure, the *PE Table*, with the same dimensions as the VRD. Each element represents a PE and contains information such as resource availability and routing data. Each row in the *PE Table* is scanned from left to right, starting from the row determined by the *Dependency table*, until a free unit is found. The *Dependency table* is then updated if the microop just placed produces a value(s). The configuration for the multiplexers that select the inputs to PEs are generated from information stored in the *PE Table*.

register	location

(a) *Dependency Table*.

col row	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

(b) *PE Table*.

Figure 5.3: Tables before first P&R.

5.3.3.1 Example P&R

To enhance the understanding of the P&R algorithm, a time-line of the P&R algorithm processing the microops in Listing 5.2 on page 81 is presented. Remember that the size of the VRD is 4×4 with one memory access per row via the rightmost PE (see chapter 4).

The state of the key data structures just before P&R of listing 5.2 starts is shown in figure 5.3. The *Dependency Table* is empty while all the elements of the *PE Table* are initialised to 0 i.e. not occupied. To simplify the example and enhance understanding the auxiliary tables necessary for emitting microops to memory are ignored. Since the *Dependency Table* is empty, the first microop for P&R cannot be data dependent on another microop. So, the microop on line 1 in listing 5.2 is placed on PE$_{00}$. Cell 00 of the *PE Table* is changed to 1 indicating that a microop has been mapped to it. Entries for %r5 and %f are made in the *Dependency Table* as %r5 and %f are defined in line 1 of Listing 5.2. To help in tracking flag dependencies, microops that define flag values are tagged during translation from MachInstrs to microops. This allows the P&R algorithm to know when to add the flag register to the *Dependency Table*. Figure 5.4 shows the state of PR tables just after the mapping of line 1 of Listing 5.2.

To determine where to place line 2 of Listing 5.2 a search for where the input operands are defined is made. $\%r2$ is not in the *Dependency Table* so will be supplied by the register file. However, $\%r5$ and $\%f$ are defined by PE$_{00}$ requiring that the microop be placed on row 1 onwards. Next, the PE table is searched, starting from row 1 and moving from left to right, for a free PE. PE$_{01}$ is the

register	location
$\%r5$	PE_{00}
$\%f$	PE_{00}

(a) *Dependency Table.*

col row	0	1	2	3
0	1	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

(b) *PE Table.*

Figure 5.4: Tables after first P&R.

register	location
$\%r5$	PE_{00}
$\%f$	PE_{00}
$\%r3$	PE_{01}

(a) *Dependency Table.*

col row	0	1	2	3
0	1	0	0	0
1	1	0	0	0
2	0	0	0	0
3	0	0	0	0

(b) *PE Table.*

Figure 5.5: Tables after second P&R.

first empty PE, so the microop is mapped to it. An entry for $\%r3$ is added to the *Dependency Table* as $\%r3$ is produced by the microop. Cell 01 is then set to 1. Figure 5.5 shows the tables after line 2 of listing 5.2 has been placed and routed.

The same process used in the mapping of line 1 and line 2 is employed for line 3 and line 4. The state of the *Dependency Table* and *PE Table* after placing and routing each line is shown in figure 5.6 and figure 5.7. Note that the microop in line 4 is placed on the rightmost PE as only the rightmost PE can access memory.

5.3.3.2 Handling Register False Dependencies

Running the RIG pass post register allocation allows the seamless intermixing of standard and reconfigurable instructions. However, this introduces false dependencies (because of limited registers in the CPU) among microops which must be eliminated to exploit fully the processing resources available on the VRD. This section shows that the P&R algorithm is capable of handling false dependencies introduced by register allocation. The microops in listing 5.3 on page 85 have a

register	location
$\%r5$	PE_{00}
$\%f$	PE_{00}
$\%r3$	PE_{01}
$\%t1$	PE_{02}

(a) *Dependency Table.*

col row	0	1	2	3
0	1	0	0	0
1	1	0	0	0
2	1	0	0	0
3	0	0	0	0

(b) *PE Table.*

Figure 5.6: Tables after third P&R.

register	location
$\%r5$	PE_{00}
$\%f$	PE_{00}
$\%r3$	PE_{01}
$\%t1$	PE_{02}

(a) *Dependency Table.*

col row	0	1	2	3
0	1	0	0	0
1	1	0	0	0
2	1	0	0	0
3	0	0	0	1

(b) *PE Table.*

Figure 5.7: Tables after fourth P&R.

number of false dependencies that could prevent their parallel execution.

```
1      %r1 = add %r3, %r2
2      %r3 = sub %r9, %r2
3      %r6 = and %r3, %r5
4      %r3 = add %r8, %r2
```

Listing 5.3: Microops with false dependencies.

There is a false dependency (Write After Read (WAR)) between between line 1 and line 2 of Listing 5.3. The *sub* and the *add* cannot execute in parallel because of register constraints. A false dependency, Write After Write (WAW)—introduced by the register allocator—exists between line 2 and line 4. They write to the same register, therefore cannot be executed in parallel.

Figures 5.8 to 5.11 on pages 86 to 87 show how the P&R algorithm eliminates false dependencies. Figure 5.8 depicts the state of the *Dependency table* and the *PE*

register	location
$\%r1$	PE_{00}

(a) *Dependency Table.*

row \ col	0	1	2	3
0	1	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

(b) *PE Table.*

Figure 5.8: Tables after placing line 1.

register	location
$\%r1$	PE_{00}
$\%r3$	PE_{01}

(a) *Dependency Table.*

row \ col	0	1	2	3
0	1	1	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

(b) *PE Table.*

Figure 5.9: Tables after placing line 2.

table after line 1 is placed. From the *Dependency table* there is no true dependency between the *add* and the *sub*, therefore line 2 is placed in the same row as line 1 (see figure 5.9). The *and* is placed on the second row as it is dependent on line 2 (see figure 5.10). There is no true dependency between line 2 and line 4, so line 4 is placed in cell 00 of figure 5.11. The entry for %r3 in the *Dependency table* of figure 5.11 now refers to the most recent position where %r3 is produced, hence the false dependency is eliminated.

5.3.3.3 Handling False Memory Dependencies

Unlike registers, tracking data dependencies in memory, at compile time, is difficult and expensive as memory addresses are often computed dynamically. Disambiguation or alias analysis, telling whether two memory references access the same memory location, is further complicated by the use of pointers. The P&R algorithm, therefore, is ‘conservative’ when reordering memory accesses.

The P&R algorithm can move a load before another load as long there is no

register	location
$\%r1$	PE_{00}
$\%r3$	PE_{01}
$\%r6$	PE_{10}

(a) *Dependency Table.*

col row	0	1	2	3
0	1	1	0	0
1	1	0	0	0
2	0	0	0	0
3	0	0	0	0

(b) *PE Table.*

Figure 5.10: Tables after placing line 3.

register	location
$\%r1$	PE_{00}
$\%r3$	PE_{02}
$\%r6$	PE_{10}

(a) *Dependency Table.*

col row	0	1	2	3
0	1	1	1	0
1	1	0	0	0
2	0	0	0	0
3	0	0	0	0

(b) *PE Table.*

Figure 5.11: Tables after placing line 4.

intervening store. The intervening store restriction prevents reading a memory location that has not be written; at compile time the P&R algorithm cannot guarantee that the load and store do not alias.

5.4 Summary

This chapter discussed the design of the DCE which is responsible for dynamically mapping kernels onto VIREMENT's reconfigurable hardware. The logical steps required to map a function presented in LLVM IR format to executable binary were presented and discussed. Examples were used to enhance understanding of the process. The simple, novel P&R algorithm, which is the heart of theDCE, was also presented. Using examples, the P&R algorithm is shown to be able to handle false data dependences introduced by making the microops generation post pass which allows the seamless intermixing of microops and standard instructions. How the algorithm handles false memory dependencies is discussed last.

The next chapter is on the design of VIREMENT's evaluation framework. The framework allows the gathering of system-level performance estimates.

6 Evaluation Methodology

Replicating a hardware design in software to evaluate its performance has always been a fundamental design tool of architects. This chapter describes the methodology and framework that allows performance statistics to be gathered for VIREMENT at the system level using software-based simulation. The framework and some of the reference applications in chapter 3 are used, in the next chapter, to qualitatively determine if VIREMENT boosts single-thread performance efficiently.

6.1 Evaluating Reconfigurable Processors

The main aim of performance evaluation is to quantitatively assess the quality of a processor 'early' in the design process i.e. before building the actual hardware. This is often achieved by simulating system behaviour and generating quantitative estimates of expected performance. Relying only on intuition and experience is no longer adequate for estimating performance due to the complexity of modern mobile processors. In a reconfigurable processor, such as VIREMENT, performance is largely determined by these key factors: the architecture of the host CPUs and the reconfigurable unit, the system integration mechanism and the applications to be executed. A good simulator should model all these factors.

Pure computational performance, measured via the execution time of a reference application, is often the primary concern of architects. For mobile processors, however, area and power consumption are important. Presently, the framework lacks area and performance estimation capabilities. As such, area and performance estimation are viable extensions to the framework. Possible starting points are discussed in chapter 8.

Designing and building a Reconfigurable Instruction Set Processors (RISP) simulator largely involves integrating a model of the reconfigurable unit into a regular CPU performance simulator (in this case gem5 [104, 105]). Simulation of the reconfigurable unit falls into two broad categories: functional simulation (this work's method of choice) and execution-based co-simulation. Each method is discussed below.

6.1.1 Functional Simulation

Functional simulation avoids explicit modelling of the reconfigurable hardware by treating configurations (only one particular configuration is used for a fine-grained reconfigurable hardware) of the reconfigurable unit as additional CPU instructions. A static function and timing behaviour, often obtained from a gate level model, is assigned to these additional instructions. Functional simulation is often used for RISPs [51, 52, 106] since the reconfigurable hardware is just another functional unit in the CPU that gets its operands from the CPU's register file.

Functional simulation is implemented by extending a CPU performance simulator with an additional software component that implements the functional and timing behaviour of reconfigurable unit. During simulation, the CPU performance simulator steers reconfigurable operations to the new component.

6.1.2 Execution-Based Co-simulation

In execution-based co-simulation, the reconfigurable unit is not abstracted but is simulated in its own cycle-accurate simulation environment usually interfaced with a cycle-by-cycle CPU simulator. A particular configuration is simulated by loading the configuration into the model and executing the configuration. Models are very detailed, covering the reconfigurable hardware, the configuration and the control circuitry. As such, they are often written at the Register Transfer Level (RTL) level in Hardware Description Languages (HDLs), such as Verilog

[107] and simulated with cycle-accurate HDL simulators, such as Synopsys VCS [108]. The HDL simulators export interfaces that provide CPU simulators with direct access to the simulation kernel.

Execution-based co-simulation produces more accurate performance estimates compared to functional simulation but developing and testing such detailed models is non-trivial and expensive. As such, the evaluation framework uses functional simulation for stimulating the VRFU.

6.2 VIREMENT Evaluation Framework

This section introduces VIREMENT's evaluation framework. The framework, based on the gem5 [104, 105] simulator which allows for the gathering of system level performance estimates.

VIREMENT is particularly suited for functional simulation as it is a RISP. Using an execution-based approach may produce more accurate results but the overhead of communicating and synchronizing across two different simulation environments reduces simulation speed unacceptably [109] especially when modelling CMP designs. The simulator, therefore, trades accuracy for speed by employing functional simulation for the VRFUs.

6.2.1 Major Framework Components

The framework consists of:

- **Gate-level Model of the VRFU:** This describes the function, timing, and structure of the reconfigurable functional unit (see chapter 4) in terms of structural interconnection of Boolean logic blocks. This is used, with synthesis tools, to obtain latency and area estimates. Latency estimates are used to calibrate simulation models.

- **Architecture Simulator:** This consists of ISA models, CPU models and other components that, together, enable the execution of (standard and reconfigurable) instructions and gathering of performance statistics. The architecture simulator leverages gem5.

- **Compilation System** This compiles the reference applications (chapter 3) for the simulator. The main component is an implementation of the Dynamic Compilation Engine (DCE) (see chapter 5). The compilation system relies largely on LLVM compiler infrastructure.

6.2.2 Gate-Level Model

The 4 × 4 VRFU in chapter 4 is described in Verilog [107] and synthesised—with Design Compiler [108] for the NanGate 45 nm standard-cell library [110]—to obtain area and latency estimates.

Designing a functional model as described in section 6.1 requires the latency of VRFU relative to the CPU. This is obtained by synthesising a Verilog description of VIREMENT's host CPU (see page 60). The CPU's Verilog description builds on MARS [111], a ARM-like processor. Timing estimates from the gate-level model is used to derive the latency of the VRFU relative to the CPU. CPU synthesis also targets the NanGate 45 nm standard-cell library [110].

6.2.3 Architecture Model and Simulator

The performance simulator is based on gem5 [104, 105]. The key features of gem5 are summarised below:

- **Object-Oriented Design:** Major simulation structures (CPUs, buses, caches, etc.) are represented as objects. These objects are composed to describe the VIREMENT architecture.

- **Discrete-Event Simulation Core:** A simulation is basically a collection of objects (CPU, cache, etc.) interacting directly through method calls. Each of these objects is responsible for managing its own timing by scheduling its own events on the global event queue. The level of granularity within each object is independent of others.

- **Full System Multicore simulation:** gem5 simulator models Realview's ARM development board with sufficient detail to boot unmodified Linux 2.6.35+

with up to 4 CPUs. The OS required no modification as the VRFU is completely transparent at the system level.

6.2.3.1 CPU Objects

The CPU object(s) is cycle-approximate. Cycle-approximate, in modelling taxonomy, refers to models that promise that operations will generally take the correct number of cycles [112]. However, not all cycles within each operation are modelled, unlike cycle-accurate models. For instance, a 64-bit multiply operation with a latency of, say, 6 clock cycles may only produce the final result expected at the end of 6 cycles i.e. it only gives the final value without providing intermediate values. This differs from cycle-accurate models that compute and present each intermediate value. As such, exactness is traded for simulation speed in the framework. Very few cycle-accurate performance models exist, most simulators are cycle-approximate [113].

In gem5, the ISA models plug into the generic CPU object and the memory systems without having to specialize one for the other. The CPU object and ARM ISA model were extended to model microops and the VRFU.

6.2.4 Compilation and Execution Model

VIREMENT targets forward-scaling programming environments where kernels are dynamically compiled (see chapter 1). To enhance understanding, a forward-scaling programming environment (Intel ArBB [114]) is described next.

6.2.4.1 A Forward-Scaling Programming Environment

Intel ArBB [114], depicted in figure 6.1 on page 94, is an example of state-of-the-art forward-scaling programming systems. The framework's compilation system models the ArBB (see next section). Kernels (parallel regions) are expressed in ArBB language, an embedded language—as types, control flow constructs, and operators are all expressed using standard C++ constructs—whose syntax is implemented as an Application Programming Interface (API). The kernels are complied with a standard C++ complier, linked with the ArBB library and distributed as

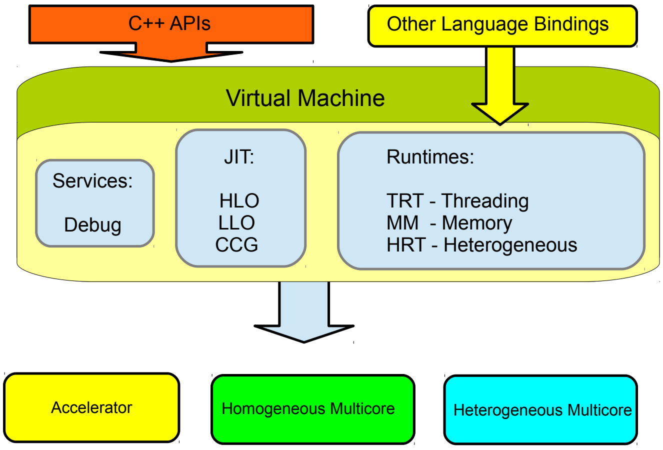


Figure 6.1: ArBB Virtual Machine.

normal applications. Function calls to kernels at runtime trigger mapping and compilation.

The ArBB VM dynamically maps the abstract, latent parallelism in kernels, expressed with the API, onto parallel mechanisms (e.g. threads, SIMD instructions, etc.) in the physical machine. As such, it provides four major services:

- The Threading Runtime (TRT) dynamically selects the task granularity and synchronization method to suit the underlying architecture. This is a key to forward scaling as threading and synchronization overhead change between processor generations. The TRT provides a fine-grained threading model for both data and task parallel threading.

- The Heterogeneous Runtime (HRT) orchestrates the loading and execution of code on accelerators. It is responsible for moving data between the host processor and the accelerators.

- The Memory Manager (MM) manages the memory with kernels in different logical memory space. It has lock-free dynamic memory allocators as well as a reference-counting garbage collector. It is responsible for allocation, data formatting and partitioning data, in conjunction with the TRT and the HRT, for parallel operations.

- JIT compiler dynamically builds intermediate representations of the computation specified by the ArBB API, performs optimisations and generates

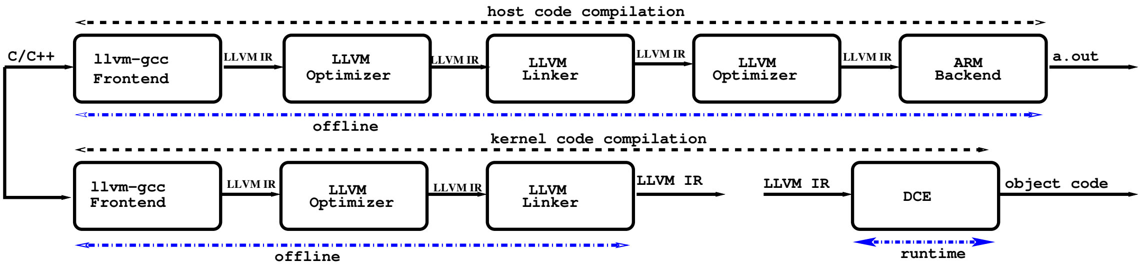


Figure 6.2: Framework's compilation system.

object code for execution. Code for a kernel is compiled lazily and is cached for reuse on subsequent calls. The complier is split into three components: High-Level Optimizer (HLO), Low-Level Optimizer (LLO) and Converged Code Generator (CCG).

6.2.4.2 Compilation on the Framework

The framework's application compilation and execution system (depicted in figure 6.2) mimics the ArBB (described earlier) which is an example state of the art forward-scaling programming environment. As such, it divides an application into two parts: kernels and host. Kernels are those functions in an application targeting the VRFU while the rest (and often bulk) of the application is the host.

Kernels are complied from source to LLVM IR—the persistent, offline variant—with llvm-gcc [115]. llvm-gcc is a derivative of the GNU Compiler Collection (GCC) [116] with GCC's optimizers and code generators replaced with those from LLVM. The host, rewritten to reference the kernels through the DCE like in forward-scaling programming environments, is complied to object code for the target CPU. The kernels, now in LLVM IR, are embedded into the host's object code and distributed as normal application binaries.

At runtime, a reference to a kernel triggers its compilation, using the DCE, and execution on the VRFU. Kernels are complied once and cached for subsequent use. Performance statistics in the evaluation framework (presented in the next chapter) are only gathered across kernels since the aim is to evaluate single thread performance of execution on the VRFU.

The DCE is based on LLVM Compiler Infrastructure, release 2.8. Development and testing was largely on a virtual platform based on Open Virtual Platform

(OVP) [117]. A virtual platform is a functional software model of a full system used for software development in the absence of hardware [118]. OVP is an instruction cycle accurate simulator, in which events are specified in terms of the processing of an instruction stream, making it much faster than the cycle approximate evaluation framework.

6.3 Summary

The simulation methodology and the design and implementation of the simulation infrastructure was the focus of this chapter. Two common simulation methods for reconfigurable processors, functional simulation and execution-based co-simulation were discussed. Functional simulation is more suited to VIREMENT (and faster); hence its adoption for the evaluation framework. The CPU model used in simulation was calibrated using gate level models. It was also shown that the framework's compilation system models forward-scaling parallel programming environments that VIREMENT is targeting.

The next chapter uses the framework and some of the reference applications in chapter 3 to qualitatively determine if VIREMENT boosts single-thread performance efficiently.

7 Results and Analysis

7.1 Introduction

Previous chapters introduced VIREMENT and its associated JIT complier DCE, a mobile CMP platform built to accelerate single threads using Reconfigurable Hardware (RH). This chapter quantitatively evaluates the performance of the platform, via simulation, to determine how well it meets its stated objective: the efficient acceleration of single threads in dynamically generated code. Simulations are based on the methodology and framework of chapter 6.

7.2 Area

Area, along with power/energy consumption, is an important constraint for mobile processors often used in portable devices such as smart-phones. Mobile processors are particularly sensitive to cost [119] and area is directly related to cost—cost per die is roughly related to the square of die area [8].

Table 7.1 on page 98 shows the area of a VIREMENT core for different Reconfiguration Cache (RC) capacities. The area estimates are from gate level models as described in chapter 6. `BC` is the baseline core and is similar to the cores in VIREMENT but without the VIREMENT Reconfigurable Functional Unit (VRFU) (see chapter 4). `VIREMENT_`x refers to the baseline core extended with a VRFU having a RC capacity of x configurations. As such, `VIREMENT_4` is a VIREMENT core with RC of 4 configuration slots. The number of gates is based

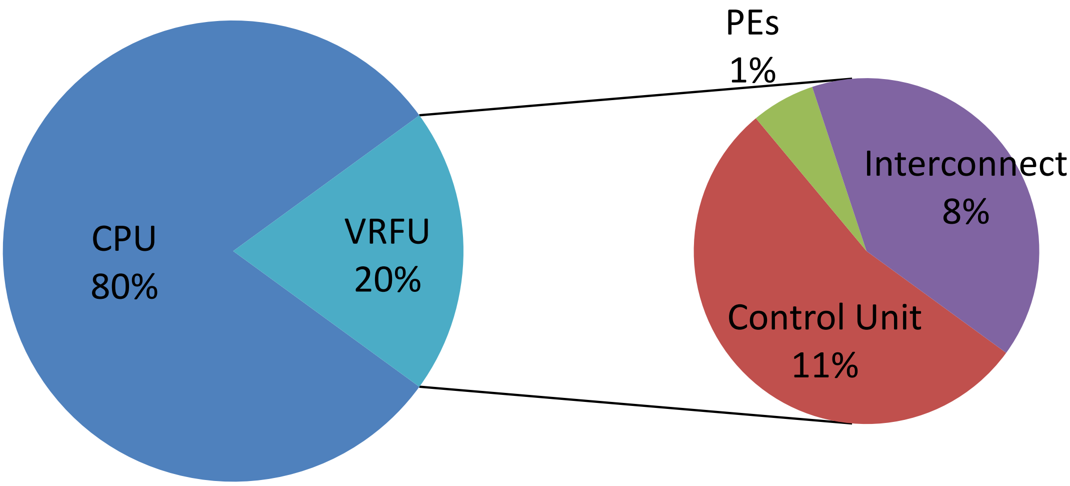


Figure 7.1: VIREMENT's resource usage.

on the assumption that designs are implemented using AND2_X1 (2-input-and gate with driving strength of 1) gates. Area overhead is the ratio of the area of `VIREMENT_`x to that of `BC`.

Design	Area (µm^2)	Number of gates	Area overhead
`BC`	297 920	280 000	1.00
`VIREMENT_4`	366 951	344 878	1.23
`VIREMENT_8`	368 185	346 039	1.24
`VIREMENT_16`	372 538	350 130	1.25
`VIREMENT_32`	381 984	359 008	1.28

Table 7.1: VIREMENT area cost.

From table 7.1, a VIREMENT core increases the area of the baseline core by only 23% – 28%. Figure 7.1 shows the percentage make up of `VIREMENT_16` core with the VRFU split into there components: *Control Unit*, *PEs* and *Interconnect*. *Control Unit* is the VIREMENT Control Unit (VCU) (see page 71) while *PEs* refers to the VIREMENT Reconfigurable Datapath (VRD) (see page 63) without the interconnection network (see page 66) which is reported separately as *Interconnect*.

The majority of the VRFU, which is 20% of the `VIREMENT_16` core, is the interconnection network and the control unit. The PEs accounts for just 1% of the core's area. The interconnect is bigger than the computational elements because it provides each PE with access to all general purpose registers and the status register in the host CPU. The size of the interconnect can be reduced by providing only a subset of the registers in the VRFU. This, however, makes code generation more difficult and hits performance as the complier is now required to move data to and from this subset of registers before and after processing on the VRFU.

7.3 Power

Power is another critical design constraint in mobile processors as they often run on batteries without active cooling (most of the power consumed by a CMOS circuit is converted into heat). There is no physical implementation of the VIREMENT presently to take power measurements. Gate or netlist level power estimation techniques [120, 121, 122, 123], however, can be used to obtain accurate power estimates [121, 122].

Power estimates are obtained using the Nangate 45nm cell library [110], the Synopsys Power Complier [108] and VIREMENT's gate-level netlist (see page 92). Power Complier's power model [124] requires the switching activity of each net to calculate power. Switching activity is the static probability that a signal is at a particular logic state plus the toggle rate (the number of logic-0-to-logic-1 and logic-1-to-logic-0 transitions per unit of time). It is obtained most often from simulations but this requires simulation vectors that represent actual application behaviour. It is difficult, however, to verify that the test cases covered are the ones that cause the highest power consumption [125]. Hence, power is estimated using the vectorless technique [121, 125]. This involves annotating the VIREMENT's primary inputs with the worst-case switching values (in this case toggle rates of 80%) which are propagated throughout the design using Power complier's BDD-based probabilistic estimation algorithm [120]. This approach has been shown to provide realistic power-consumption estimates [125].

The worst-case power consumption for the entire `VIREMENT_16` is estimated at 60 mW. 55% is consumed by the CPU and the rest by the VRFU (see figure 7.2

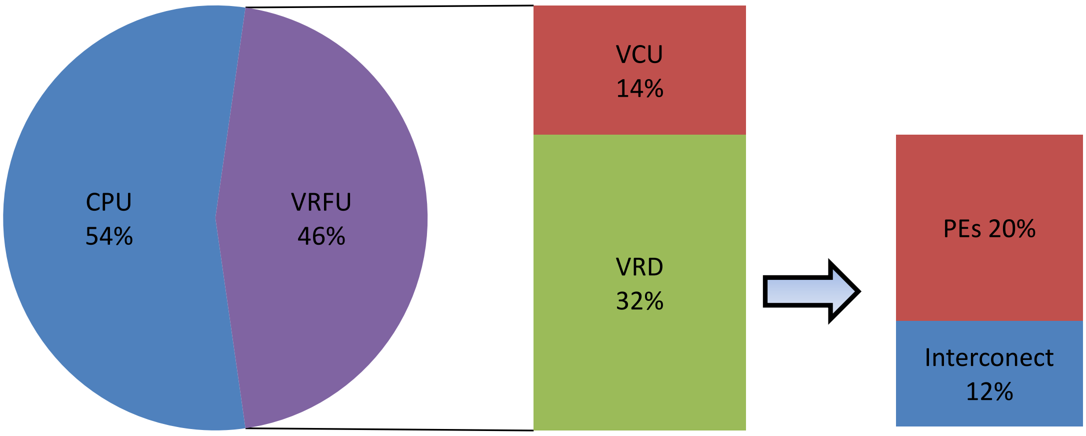


Figure 7.2: VIREMENT_16 worst-case power consumption breakdown.

on page 100). The worst-case power consumption in the VRFU is estimated to be 27 mW with 70% consumed by the VRD and the rest by the VCU. Most power in the VRD is consumed by the PEs as shown in the figure.

With a worst-case power consumption of only 60 mW for a VIREMENT_16 core, a reasonable number of VIREMENT cores can be integrated into a mobile SoC with room for other SoC components without exceeding the power limit of 2–3 W for mobile phones.

The power consumed by DCE during compilation is assumed to be negligible as compilation is usually only a tiny fraction of execution time as shown in table 7.11 on page 117.

7.4 Performance

This section evaluates the performance advantage of VIREMENT. Execution time on VIREMENT_x is compared to that on BC. Only integer dominated applications (bfs, fib, nqueens, pathfinder, sort) from those characterised (see chapter 3) are evaluated as VIREMENT lacks support for floating point operations (see chapter 4). x264 and freqmine are integer dominated but fail to run on the evaluation framework because of compilation issues. Further, measurements are across kernels as explained in chapter 6. Applications for VIREMENT were prepared as described in chapter 6. The serial version of each application is used for the single processors but measurement is still restricted to the kernels that are parallized in the multi-threaded version. The dynamic instruction count across the kernels of each of the application is shown in table 7.2.

Application	Dynamic instruction count
`bfs`	2.5×10^{8}
`fib`	2.2×10^{12}
`nqueens`	1.7×10^{13}
`pathfinder`	3.0×10^{9}
`sort`	2.6×10^{9}
`average`	3.8×10^{12}

Table 7.2: Dynamic instruction count across kernels.

Table 7.3 shows the speedup (the ratio of execution time on `BC` to that on `VIREMENT_`x) for `VIREMENT_16` with 1, 2 and 4 cores. The table lists, per application, the speedup for 1-core ($1xCPU$), 2-core ($2xCPU$), 4-core ($4xCPU$) `VIREMENT_16`; Arith_Mean (the arithmetic mean of speedup across the three configurations of `VIREMENT_16`) and Geo_Mean (the geometric mean of speedup across the three configurations of `VIREMENT_16`).

From the table, `VIREMENT_16` outperforms the baseline by an average of $2.6\times$ for the 1-core, $2.1\times$ for the 2-core and $1.9\times$ for the 4-core. `pathfinder` benefits the most with speedups of $3.7\times$, $2.8\times$ and $2.5\times$ on the 1-core, 2-core and 4-core configuration.

Performance on VIREMENT is largely determined by the interplay of three factors: (1) the Instruction Level Parallelism (ILP) within each basic block, (2) the number of VRFU mappable operations within each basic block and (3) the relative execution weight of each basic block. As such, an ideal application for acceleration on VIREMENT is one whose frequently executing basic blocks contain a high number of independent operations that can be executed on the VRFU. `pathfinder` is nearer the ideal than the other applications. Applications with relatively low ILP like `bfs` (see page 48) still benefit as the two dimensional structure of the VRD

Application	Speedup (Execution Time BC/Execution Time VIREMENT_16)		
	1xCPU	2xCPU	4xCPU
bfs	3.0	2.1	1.6
fib	1.9	1.9	1.9
nqueens	2.5	1.8	1.8
pathfinder	3.7	2.8	2.5
sort	1.7	1.7	1.7
Arithmetic_Mean	2.6	2.1	1.9
Geometric_Mean	2.5	2.0	1.9

Table 7.3: VIREMENT_16 speedup.

executes strings of dependent operations faster than the host CPU (see chapter 4).

From table 7.3 on page 102, speedup drops when moving from the serial version of an application to the threaded. Parallel programs use synchronization primitives for controlling the interactions of threads and avoiding race conditions. These primitives can't always be mapped to the VRFU; they are often in system libraries or use operations not supported on the VRFU, hence the decrease in speedup. The relative time spent in these synchronization primitives often increases as the number of threads increase. As such, performance on VIREMENT relative to the baseline tends to decrease as the number of threads increase.

Speedup per application for VIREMENT_4, VIREMENT_8 and VIREMENT_32 is presented in tables 7.4 to 7.6 on pages 103 to 104. Tables 7.3 to 7.6 show that some applications are more sensitive, in terms of performance, to the capacity of the RC than others. Sensitivity is largely related to the number of configurations in each application. fib has only 1 configuration (see table 7.7 on page 104), hence changing the cache capacity from 4 to 32 configurations does not affect performance. sort, on the other hand, with 59 configurations is very sensitive to the capacity of the RC.

Application	Speedup (Execution Time BC/Execution Time VIREMENT_16)		
	1xCPU	2xCPU	4xCPU
bfs	0.43	0.25	0.15
fib	1.9	1.9	1.9
nqueens	2.4	1.8	1.8
pathfinder	3.6	2.8	2.4
sort	1.5	1.5	1.5
Arithmetic_Mean	2.0	1.5	1.5
Geometric_Mean	1.6	1.1	1.1

Table 7.4: VIREMENT_4 speedup.

Application	Speedup (Execution Time BC/Execution Time VIREMENT_16)		
	1xCPU	2xCPU	4xCPU
bfs	3.0	2.1	1.6
fib	1.9	1.9	1.9
nqueens	2.5	1.8	1.8
pathfinder	3.7	2.8	2.4
sort	1.5	1.5	1.5
Arithmetic_Mean	2.5	2.0	1.9
Geometric_Mean	2.4	2.0	1.8

Table 7.5: VIREMENT_8 speedup.

7.5 Area and Power Efficiency

From section 7.2 and section 7.4, it is evident that performance increase is at the cost of significant area and power overheads. An important question is whether a conventional processor could have been more efficient. This is answered with the help of Pollack's Rule [126] which states that the performance increase, by microarchitecture alone, is roughly proportional to square root of increase in

Application	Speedup (Execution Time BC/Execution Time VIREMENT_16)		
	$1xCPU$	$2xCPU$	$4xCPU$
bfs	3.0	2.1	1.6
fib	1.9	1.9	1.9
nqueens	2.5	1.8	1.8
pathfinder	3.7	2.8	2.5
sort	1.7	1.7	1.7
Arithmetic_Mean	2.6	2.1	1.9
Geometric_Mean	2.5	2.0	1.9

Table 7.6: VIREMENT_32 speedup.

Application[a]	Number Configurations	CPU Instrs./Configuration[b]
bfs	31	6.8
fib	1	7.0
nqueens	12	5.5
pathfinder	27	3.7
sort	59	3.2

[a] Serial version.

[b] Weighted average of CPU instructions per configuration.

Table 7.7: Number of instructions per configuration.

complexity. As such, doubling the number of gates in a conventional processor delivers only 40% more performance on the same technology node.

Pollack's Rule, formulated for desktop processors, is applicable to modern mobile processors as they both use the same micro (-architectural) techniques as evidenced by figure 1.1 on page 17.

It is assumed that the BC of section 7.4 can be enhanced to create a core, $\mathtt{BC}_{seq}$, with greater sequential performance than that of BC by using the resources of multiple BCs. Let the performance of a BC core be 1. Architects can expend the resources of r BCs to create a more powerful $\mathtt{BC}_{seq}$ with single thread performance:

From Pollack's Rule:

$$perf(r) = \sqrt{r} \tag{7.1}$$

With equation (7.1), adding r BC more resources increases sequential performance by $\sqrt{r}$. So, performance can be doubled at a cost of four BCs, tripled for nine BCs, and so on. Note that equation (7.1) only accounts for cores and does not factor in components such as last-level caches, the on-chip interconnect, etc.

The area overheads for VIREMENT, see table 7.1, can be written in terms of r BCs. For instance, the 1-core VIREMENT_16 uses 1.25 BCs to increase pathfinder's performance by 3.5× relative to the BC. Since the area of VIREMENT and that of the conventional sequential processor can be expressed in r BCs, the ratio of their areas can be calculated.

Table 7.8 on page 105 is the area ratio of $\texttt{BC}_{seq}$ to VIREMENT_16 for 1-, 2- and 4-core configurations. In other words, it is the additional area that a conventional core needs (according to Pollack's Rule) to match the performance of VIREMENT_16. For multiple cores, it is assumed that loads across the cores are perfectly balanced.

Application	$Area\,\texttt{BC}_{seq}/Area\,\texttt{VIREMENT\_16}$		
	$1xCPU$	$2xCPU$	$4xCPU$
bfs	7.1	3.6	2.1
fib	3.0	3.0	5.0
nqueens	5.0	2.3	2.6
pathfinder	11	6.4	4.8
sort	2.3	2.3	2.3
Arithmetic_Mean	5.6	3.6	3.0
Geometric_Mean	4.8	3.3	2.8

Table 7.8: VIREMENT_16 area efficiency.

From table 7.8, it is clear that VIREMENT is more efficient at utilizing resources to increase single-thread performance compared to a conventional design. For instance, a single (conventional) $\texttt{BC}_{seq}$ core processor running pathfinder is predicted to need nearly 1000% more resources than VIREMENT_16 to match

VIREMENT_16's performance.

Improvement in single-thread performance of conventional processors is often accompanied by an increase in power consumption. A simple power estimation [127], equation (7.2), can be used to model the relationship between power consumption and performance of such processors.

$$power = perf^{\mu} \tag{7.2}$$

μ has been estimated to be 1.75 in conventional processors [128]. With Pollack's Rule, equation (7.2) leads to equation (7.3).

$$\begin{aligned} power &= perf^{\mu} \\ &= (\sqrt{r})^{\mu} \\ &= r^{\mu/2} \end{aligned} \tag{7.3}$$

Therefore, the power consumption of a sequential microprocessor relative to a BC can be estimated with equation (7.3).

Table 7.9 shows the additional power consumed by $\mathtt{BC}_{seq}$ relative to BC for its single-thread performance to match that of VIREMENT_16. From the table, VIREMENT_16's power efficiency— and by extension energy efficiency since VIREMENT_16 computes with less power— is better than that of a conventional processor.

Application	$Power\ \mathtt{BC}_{seq}/Power$ VIREMENT_16		
	$1xCPU$	$2xCPU$	$4xCPU$
bfs	3.7	2.1	1.3
fib	1.7	1.7	1.7
nqueens	2.7	1.5	2.6
pathfinder	5.3	3.4	2.6
sort	1.4	1.4	1.4
Arithmetic_Mean	3.0	2.0	1.7
Geometric_Mean	2.8	2.0	1.8

Table 7.9: Extra power to match VIREMENT_16.

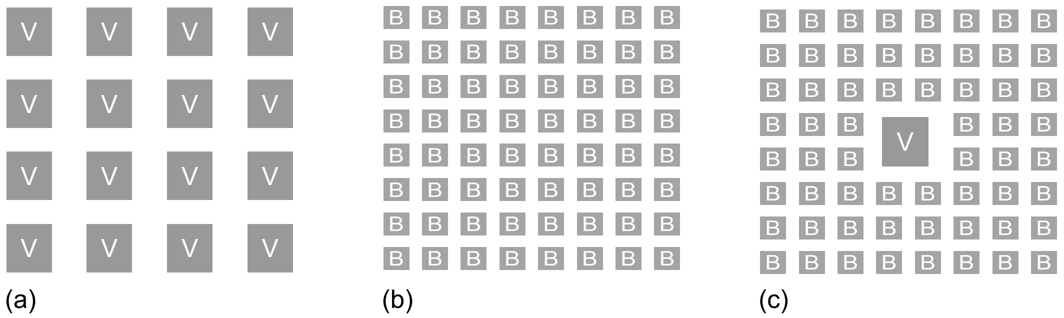


Figure 7.3: CMP design styles: (a) homogeneous CMP that replicates VIREMENT core, (b) homogeneous CMP that replicates smaller baseline core and (c) heterogeneous CMP with numerous baseline cores and one VIREMENT core as host.

In conclusion, a 'reasonable' configuration of VIREMENT, such as `VIREMENT_16`, achieves better area, power and energy efficiency compared to a conventional design for the same single-thread performance.

7.6 Power Budget Designs

The power budget of a CMP is a critical design constraint and determines the maximum number of cores in a design. This section compares, for a given power budget the performance of `VIREMENT_16`, `BC` and $\texttt{BC}_{het}$ (see figure 7.3) in terms of *performance*, *performance per watt* and *performance per joule*. $\texttt{BC}_{het}$ is a heterogeneous CMP with one VIREMENT core for executing serial sections of an application and n `BC` cores for executing parallel sections. It is assumed that the VIREMENT core is powered off during parallel execution.

7.6.1 Models for VIREMENT_16

performance, *performance per watt* and *performance per joule* calculations are based on extensions [129] of Amdahl's analytical model [10] for the theoretical maximum performance (or *performance*) in multiprocessors. Amdahl's law states that:

$$Perf = \frac{1}{(1-f)+\dfrac{f}{n}} \tag{7.4}$$

where n is the number of processors, and f $(0 \leq f \leq 1)$ is the fraction of the computation that can be parallelized. The power consumption of a `VIREMENT_16`-based CMP can be modelled with the introduction of k $(0 \leq k \leq 1)$ into equation (7.4) where it represents the power a `VIREMENT_16` core consumes while idle [129]. Assuming that a `VIREMENT_16`-based uniprocessor consumes a power of 1, by definition, the power consumed by the CMP configuration during the sequential execution phase is 1, consumed by the active core, plus $(n-1)k$ consumed by the n idling cores. The power consumption by n `VIREMENT_16` cores during the parallel execution phase is n. From equation (7.4), it takes a time $(1-f)$ plus f/n to execute the sequential and parallel sections, then the average power consumption on a `VIREMENT_16` CMP is:

$$\begin{aligned} W &= \frac{(1-f) \times \left(1+(n-1)k\right) + \frac{f}{n} \times n}{(1-f) + \frac{f}{n}} \\ &= \frac{1+(n-1)k(1-f)}{(1-f)+\frac{f}{n}} \end{aligned} \tag{7.5}$$

performance per watt (Perf/W), which represents the performance achievable at the same cooling capacity, can be modelled using equation (7.5) [129]. Since *performance* is defined as the reciprocal of execution time, this metric is essentially the reciprocal of energy. *Perf/W* of a single-core `VIREMENT_16` is 1, for a CMP composed from `VIREMENT_16` it is:

$$\begin{aligned} \frac{Perf}{W} &= \frac{1}{(1-f)+\frac{f}{n}} \times \frac{(1-f)+\frac{f}{n}}{1+(n-1)k(1-f)} \\ &= \frac{1}{1+(n-1)k(1-f)} \end{aligned} \tag{7.6}$$

Combining equations (7.5) and (7.6), *performance per joule (Perf/J)*, a metric for evaluating the performance achievable with the same battery capacity, is [129]:

$$\frac{Perf}{J} = \frac{1}{(1-f)+\dfrac{f}{n}} \times \frac{1}{1+(n-1)k(1-f)} \tag{7.7}$$

performance per joule is essentially the reciprocal of the energy delay product (performance with the same energy consumption).

7.6.2 Models for BC

The performance of a BC-based CMP relative to a VIREMENT_16 CMP can be modelled using equation (7.4) and introducing the variable s_{bc} $(0 \leq s_{bc} \leq 1)$ [129]. The variable represents the performance of BC normalized to that of VIREMENT_16. As such, a BC-based CMP's performance is:

$$Perf = \frac{s_{bc}}{(1-f)+\dfrac{f}{n}} \tag{7.8}$$

.

Similarly, BC power consumption relative to VIREMENT_16 can be modelled with two new variables: w_{bc} $(0 \leq w_{bc} \leq 1)$ and k_{bc} $(0 \leq k_{bc} \leq 1)$ [129]. w_{bc} represents active power consumption in a BC relative to that in a VIREMENT_16 while k_{bc} is the power consumption of an idle BC normalized to the same core's active power consumption. Thus, *Perf/W* and *Perf/J* for a BC-based CMP can be modelled as:

$$\frac{Perf}{W} = \frac{s_{bc}}{w_{bc}+(n-1)w_{bc}k_{bc}(1-f)} \tag{7.9}$$

$$\frac{Perf}{J} = \frac{s_{bc}}{(1-f)+\dfrac{f}{n}} \times \frac{s_{bc}}{w_{bc}+(n-1)w_{bc}k_{bc}(1-f)} \tag{7.10}$$

7.6.3 Models for BC$_{het}$

The *Pref*, *Perf/W* and *Perf/J* of a `BC`$_{het}$-based CMP, relative to a `VIREMENT_16` CMP, can be modelled, as follows, with equation (7.4) [129]:

$$Perf = \frac{1}{(1-f) + \dfrac{f}{(n-1)s_{bc}}} \tag{7.11}$$

$$\frac{Perf}{W} = \frac{1}{(1-f)\Big(1+(n-1)w_{bc}k_{bc}\Big) + \dfrac{f}{s_{bc}}\Big(\dfrac{k}{n-1} + w_{bc}\Big)} \tag{7.12}$$

$$\frac{Perf}{J} = \frac{1}{(1-f) + \dfrac{f}{(n-1)s_{bc}}} \times \frac{1}{(1-f)\Big(1+(n-1)w_{bc}k_{bc}\Big) + \dfrac{f}{s_{bc}}\Big(\dfrac{k}{n-1} + w_{bc}\Big)} \tag{7.13}$$

7.6.4 Power Equivalent Models

Assuming that a uniprocessor `VIREMENT_16` has a power budget of W_{budget} and that n_v is the maximum number of `VIREMENT_16` cores that can be instantiated on a die. Since the uniprocessor has a power consumption of 1 then the n_v cores in the CMP configuration can consume up to n_v of power. Hence, the maximum number of cores in a `VIREMENT_16` CMP can be represented as [129]:

$$n_v = W_{budget} \tag{7.14}$$

Similarly, the n_{bc} cores of a `BC` CMP consume power up to $n_{bc} \times w_{bc}$, which should be less than or equal to W_{budget}. Therefore, the maximum number cores for a budget of W_{budget} is [129]:

$$n_{bc} = \frac{W_{budget}}{w_{bc}} \tag{7.15}$$

Finally, the n_{het} cores of a BC$_{het}$ CMP consume up to $1+(n_{het}-1)w_{bc}$. So, the maximum number of cores, n_{het}, is [129]:

$$n_{het} = \frac{W_{budget}-1}{w_c} \tag{7.16}$$

7.6.5 Comparing designs

Using the analytical models, *Pref*, *Perf/W* and *Perf/J* for the three designs are evaluated and presented in figures 7.5 and 7.6 on pages 113 and 115. Figure 7.4 on page 111 shows the number of cores used in the evaluations. Note, the number of cores is rounded down when calculated with equations (7.15) and (7.16), hence the kinks in the graphs of figures 7.5 and 7.6 .

It is assumed that BC consumes 10% of full power while idling. The idle power in VIREMENT_16 is consumption in BC plus that in the VRFU. Since, the VRFU is largely a combinatorial circuit, it is assumed that leakage power dominates when it is idling. Leakage power for the VRFU, from Synopsys Power Compiler, is ≈

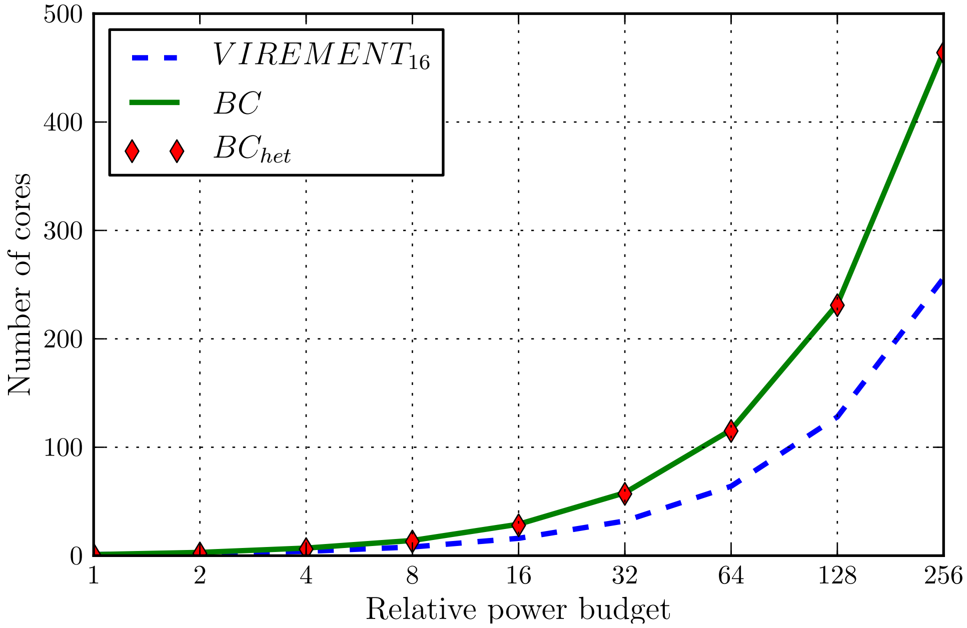
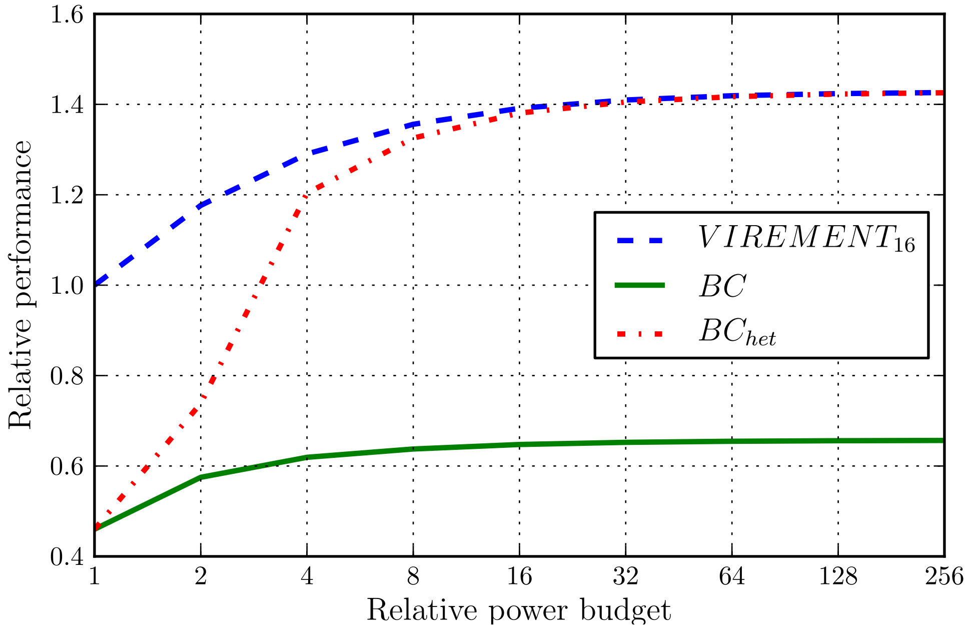
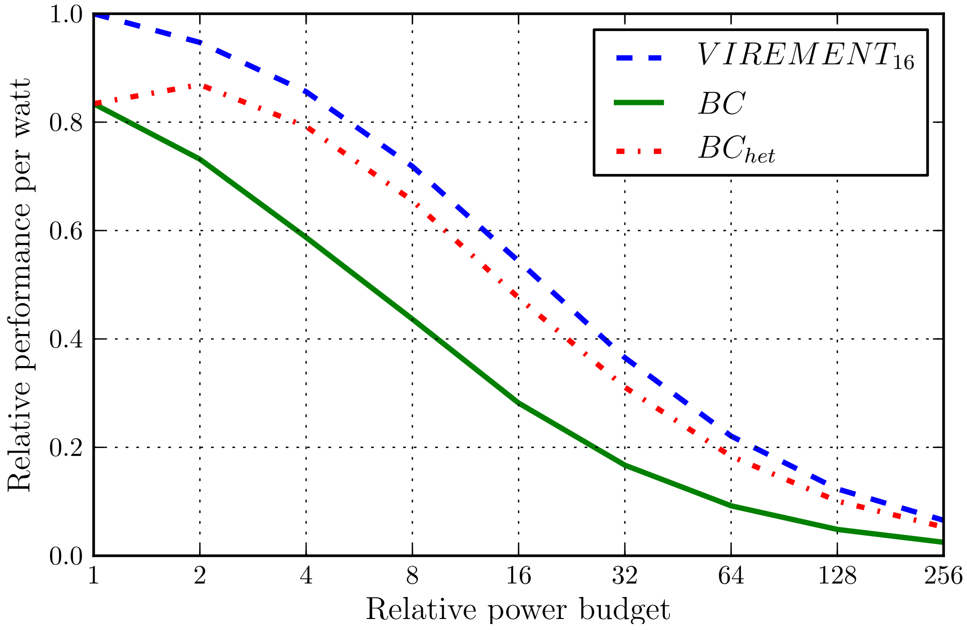


Figure 7.4: Number of cores for each power budget.

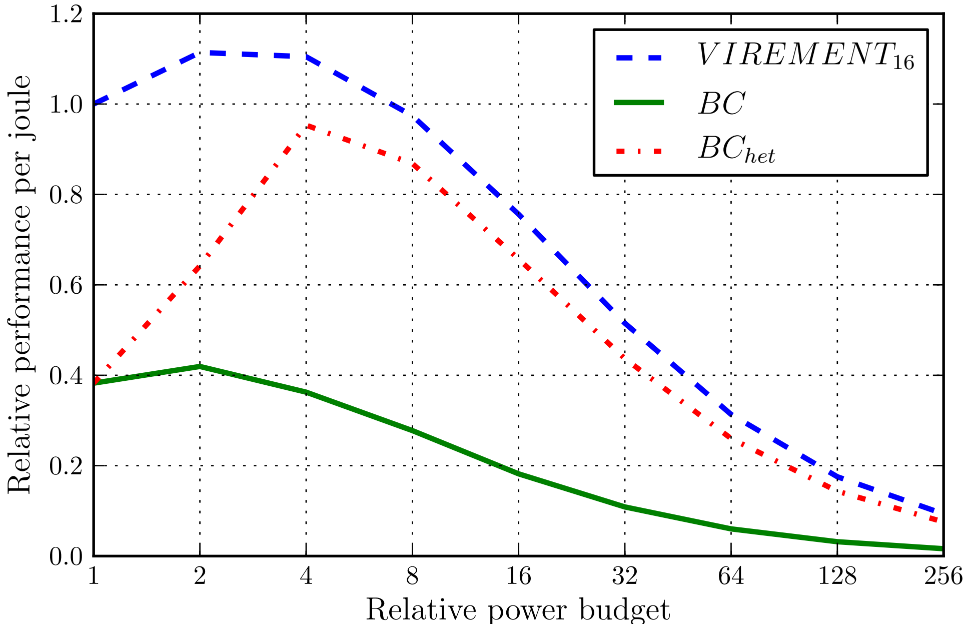


(a) performance



(b) performance per watt

Figure 7.5: Comparing designs for $f = 0.3$: (a) performance, (b) performance per watt and performance per joule.



(c) performance per joule

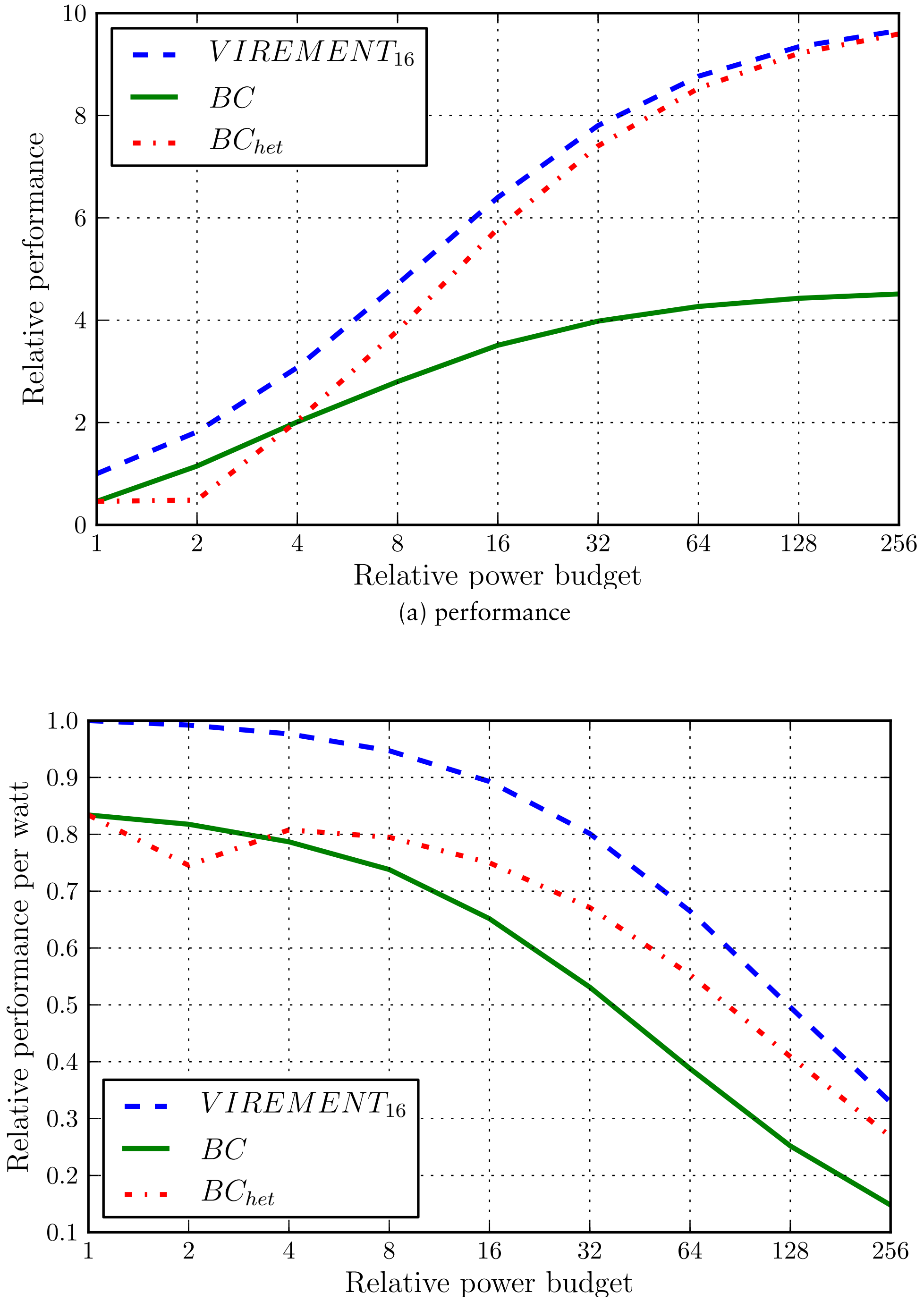
Figure 7.5: Comparing designs for $f = 0.3$: (a) performance, (b) performance per watt and performance per joule.

7% of the total power, as such, k is 0.08. s_{bc} and w_{bc} are 0.46 and 0.55 from the previous sections.

As figures 7.5a and 7.6a show, the power-equivalent *performance* of VIREMENT_16 is highest in most cases. As the relative power-budget increase, the *performance* of BC$_{het}$ approaches that of VIREMENT_16 because BC$_{het}$ can have more cores for the same power budget. BC's *performance* is the lowest across all power-budgets, even with a high f, because of BC's low single-thread capability.

Figures 7.5b and 7.6b are the power-equivalent *performance per watt* for $f = 0.3$ and $f = 0.9$. For high and low parallelism levels, VIREMENT_16 gives the best *performance per watt*. As the relative power budget increase, the *performance per watt* for the designs tends to converge. This is more pronounced when the level of parallelism is low ($f = 0.3$).

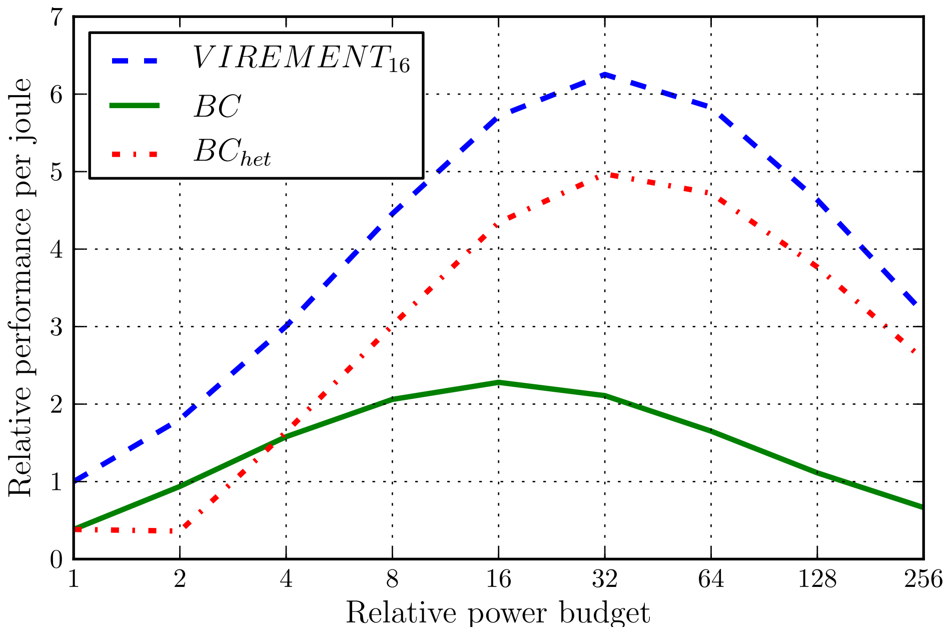
For $f = 0.3$ (see figure 7.5c), VIREMENT_16 has the highest power-equivalent *performance per joule* followed by BC$_{het}$. However, as power budget increase the



(a) performance

(b) performance per watt

Figure 7.6: Comparing designs for $f = 0.9$: (a) performance, (b) performance per watt and performance per joule.



(c) performance per joule

Figure 7.6: Comparing designs for $f = 0.9$: (a) performance, (b) performance per watt and performance per joule.

power-equivalent *performance per joule* for the three, again, tends to converge. For $f = 0.9$ (figure 7.6c), VIREMENT_16, again, has the highest *performance per joule* and then $\mathtt{BC}_{het}$. The difference in *performance per joule* between VIREMENT_16 and $\mathtt{BC}_{het}$ tends to decrease as the relative power-budget increases.

Concluding, a CMP composed from VIREMENT_16 outperforms the other alternatives, using BC cores and a heterogeneous design (combing a VIREMENT_16 core with BC cores) in terms of $Pref$, $Perf/W$ and $Perf/J$ for most power budgets and levels of parallelism. The heterogeneous design only starts to match the efficiency of VIREMENT_16 at very 'large' power budgets. Therefore, for a typical mobile processor—targeting portable devices with small power and energy budgets—VIREMENT_16 is the most appropriate design because of its performance and efficiency.

7.7 Evaluating DCE

The DCE needs to run on mobile platforms with constrained processing power, memory, and storage. For instance, every application running on Google's Android platform must be able to run on at most 130 MB of RAM and 260 MB of Flash external drive [2]. This section evaluates the suitability of DCE for resource constrained, mobile platforms, such as Android [2], using the following criteria:

1. Memory requirements of DCE since it is targeting, often, memory constrained devices such as mobile phones.

2. Translation overheads and efficiency as the DCE may have to run on battery powered devices where any computation depletes the limited battery energy.

All the applications in this section are the multi-threaded variant, running on a 2-core `VIREMENT_16` CMP.

7.7.1 Memory Requirements

The memory requirements of DCE are evaluated by measuring three characteristics: memory footprint, peak memory and the size of DCE itself. The results are presented in table 7.10. Memory footprint refers to number of unique 16-byte chunks of memory DCE references while compiling kernels. The maximum amount of memory consumed by the DCE is the peak memory and is measured in kB.

From table 7.10, all of the applications can be compiled with relatively small memory footprints. On the average, the memory footprint is about 1 MB while the peak memory used across the applications averages 410 kB. The size of DCE itself is only 6 MB (this can be reduced by 'stripping away' components of the LLVM code generator library not used in the DCE). DCE easily meets the memory requirement of mobile platforms such as Windows Phone OS 7 which requires that an application's RAM consumption must not exceed 90 MB [130].

Application	Memory Consumption	
	$Peak\,(KB)$	$Footprint\,(16B\,chunks)$
bfs	370	61 000
fib	400	53 000
nqueens	440	65 000
pathfinder	310	61 000
sort	530	74 000
average	410	63 000

Table 7.10: DCE memory consumption.

7.7.2 Translation Overheads

Mobile processors are powered by batteries making it imperative that the DCE is efficient as possible to reduce energy consumption. Table 7.11 shows the cost breakdown of execution time for all the applications evaluated in section 7.4. In the table, CPU and $VRFU$ refer to the fraction of execution time spent on the host CPU and on the reconfigurable hardware while DCE refers to the fraction of execution spent by the DCE mapping kernels to the reconfigurable hardware. From the table, translation overhead is negligible, less than 1% of total execution time.

Application	Fraction of total execution time		
	CPU	$VRFU$	DCE
bfs	0.749 2	0.210 2	0.040 7
fib	0.657 8	0.342 2	≈0.000 0
nqueens	0.715 2	0.284 8	≈0.000 0
pathfinder	0.785 9	0.210 3	0.003 8
sort	0.629 9	0.368 1	0.001 1
average	0.707 6	0.283 1	0.009 1

Table 7.11: Execution time break down

Table 7.12 shows the average number of CPU instruction needed to map a single LLVM instruction. In the table, $Kernels$ refers to the number of functions mapped to the VRFU; $LLVM\ Instrs.$ is the number of LLVM instructions in the mapped functions; $CPU\ Instrs.$ is the total number of CPU instructions used in mapping the functions and $Instrs/Trans$ is the average number of CPU instructions used to map a single LLVM instruction.

Application	Translation statistics			
	$Kernels$[a]	$LLVMInstrs.$[b]	$CPUInstrs.$[c]	$Instrs/Trans$[d]
`bfs`	2	282	13 582 401	48 164.54
`fib`	3	49	8 232 170	168 003.47
`nqueens`	3	149	13 117 990	88 040.20
`pathfinder`	2	230	14 450 358	62 827.64
`sort`	4	355	45 938 461	129 404.12
average	2.80	213	19 064 276	99 287.99

[a] Number of kernels mapped to reconfigurable unit.
[b] Number of LLVM instructions in kernels.
[c] Number of CPU instructions required for translation.
[d] Average number of instructions needed to translate one LLVM instruction.

Table 7.12: Execution time break down.

Figure 7.7 shows the average time spent in each of the major passes of the DCE while mapping the applications in table 7.2 to the reconfigurable unit. The major passes in the DCE were discussed earlier in chapter 5. From the figure, the reconfigurable instruction generation pass is only 14.50% of DCE's execution time.

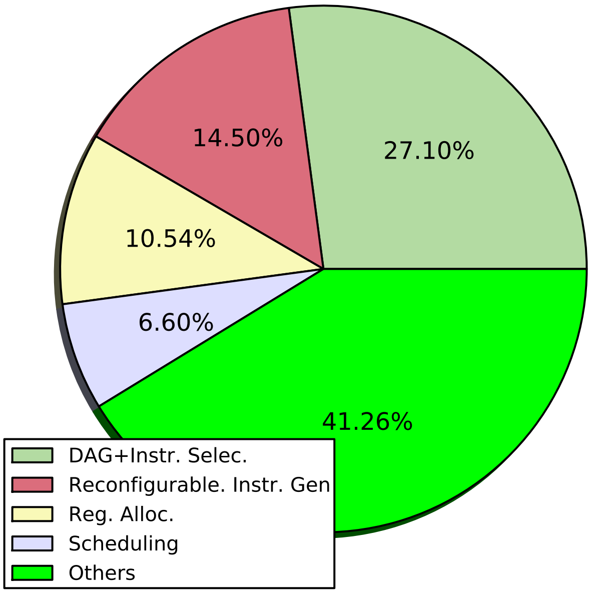


Figure 7.7: Time cost of passes in DCE.

7.8 Summary

This chapter presented the quantitative evaluation of the VIREMENT architecture and its associated JIT complier, DCE. The system increases single-thread performance by an average of 1.65× – 2.16× depending on configuration. The area and power cost of this improvement was shown to be less than that of a conventional processor design. Using analytical models a CMP based on VIREMENT was shown to outperform a conventional processor baseline in terms of *performance*, *performance/watt* and *performance/joule*. The memory requirements of the JIT complier were shown to be small.

The next chapter presents a short review of the thesis and summarises the contributions of the thesis. Suggestions on improvements are also presented.

8 Conclusions

8.1 Synopsis

Even in the present CMP era—forced upon vendors by unsustainable power consumption and ever increasing design and verification complexity—single-thread (or sequential) performance is still critical. Chapter 3 shows that the average parallel speedup across the 20 reference applications is low. Even with massively parallel applications the serial sections are a bottleneck and have a significant influence of overall performance (Amdahl's law [10]). Single-thread performance suffers as vendors move to fewer per core resources—e.g. narrower issue width, shallower pipelines, smaller out-of-order execution windows, etc.— to allow them instantiate more cores on a die. As such, single-thread performance is shown to be decreasing by figure 1.2 on page 18.

A present and future challenge is how to increase single-thread performance in CMPs **efficiently**. This challenge is more daunting for mobile CMPs, the focus of this thesis, as any proposed solution must be cost, energy and power efficient since they run fan-less from batteries and must meet a certain form factor.

Reconfigurable hardware has been used to accelerate single-thread performance in mobile processors (see chapter 2). However, modern mobile platforms are increasingly using dynamic compilation, forced upon vendors by the high cost of software development and the need for parallel applications that scale to future systems (see chapter 1). Most systems with reconfigurable acceleration require static compilation because compilation on 'traditional' reconfigurable hardware is too complex and too expensive to perform online. The few systems that map applications to reconfigurable hardware dynamically were shown, in chapter 2, to be unsuitable for resource-constrained mobile CMPs because of cost, resource or power issues.

This thesis proposed an architecture, VIrtual REconfigurable Micro-ENgine for Translation, consisting of a CPU extended with reconfigurable hardware and an associated JIT complier (the DCE), for *efficiently* boosting single-thread performance of dynamically generated code.

The reconfigurable hardware in VIREMENT is a 4×4 array of relatively simple interconnected ALUs. The design and dimensions were arrived at after profiling and analysing the reference applications in chapters 3 and 4. The interconnect is restrictive, reducing the computational intensity of the P&R algorithm, allowing the DCE to perform P&R—using a novel algorithm described in chapter 5—at runtime. The compiler pass that performs P&R in DCE is shown, in chapter 7, to be only 14.50% of the total execution time of the DCE.

VIREMENT increased performance by an average of 1.65× – 2.16×, compared to the baseline, depending on configuration. The area overhead of the reconfigurable extension was shown to be between 1.23 and 1.28. It was shown that a traditional microprocessor would need between 3.23× and 5.47× more area to match VIREMENT's performance. Such a processor would consume as much as 6× more power than VIREMENT. CMPs built from VIREMENT showed better performance, performance/watt and performance/joule compared to the other CMP design alternatives.

VIREMENT's main advantage is efficiency (area, power and energy). Efficiency is critical in VIREMENT's primary target market, mobile computers. The cost of performance improvement with VIREMENT is less than similar gain through 'traditional' means such as out-of-order execution, pielining, etc. Unlike most of the previous architectures employing reconfigurable accelerators, VIREMENT accelerates dynamically generated code. This is important as dynamic code generation is now prevalent on mobile processors. The design of VIREMENT's runtime compilation systems makes for easy integration into the various mobile system environments that employ dynamic compilation.

VIREMENT's biggest drawback is the basic-block by basic-block mapping which limits performance as the number of operations and ILP available within a basic block is small. Another drawback is the inability to perform floating point operations which have been shown to be significant in applications. The next section suggests possible ways of improving VIREMENT.

8.2 Improving VIREMENT and DCE

This section suggests possible improvements to the VIrtual REconfigurable Micro-ENgine for Translation architecture.

8.2.1 Floating Point PEs

VIREMENT was evaluated using only integer benchmarks as it lacks floating point PEs. However, most of the reference applications in chapter 3 have significant floating point operations. Adding floating point PEs, therefore, to the reconfigurable hardware would improve performance. However, floating point units are expensive, compared to integer units and are often multi-cycled. Hence, research into cost-effective method(s) of introducing such units into VIREMENT is needed. Possible directions include sharing floating point units, similar to the approach used for memory operations. Alternatively (or together with sharing), fusing two integer PEs together to form a floating point PE [131]. One of the PEs is charged with the exponent part of a floating point number while the other handles the mantissa.

8.2.2 Split compilation

Split compilation [99, 132]—a single optimisation algorithm is split into multiple compilation steps and relying on annotations, embedded in the IR, and coding conventions to transfer semantic information between the different steps—can be used to improve the quality of code generation in DCE. With this set-up, expensive but order of magnitude improving optimisations can be performed ‘online’. For instance, loop nest parallelization, expensive but order of magnitude improving, could be split into an offline and an online stage, with LLVM IR annotated to transfer semantic information between the stages. The expensive but platform independent analyses are run offline to prune the optimization space while the cheap platform specific optimisations are deferred to the online stage, when the precise execution context is known.

8.2.3 Beyond Basic Blocks

Fast, lean and efficient region mapping is needed as the current local, block-by-block, approach limits performance. Region mapping involves jointly scheduling multiple basic blocks to increase the ILP using techniques such as trace scheduling [89], superblock scheduling [88] and software loop pipeling [103]. Unfortunately, these techniques are computationally intensive making them impractical for a JIT complier, especially one on a resource constrained platform. However, with split compilation, discussed earlier, such techniques may be usable on a JIT.

8.2.4 Superscalar Processors

The design of reconfigurable unit assumes a CPU with a scalar, in-order pipeline. Mobile processors, however, are increasingly superscalar with speculative out-of-order execution. Upgrading VIREMENT for such cores is desirable. First, however, an efficient technique(s) is needed to handle synchronisation between the reconfigurable functional unit and the other functional units since in such cores the register file may not always have the up-to-date data.

8.3 Improving the Evaluation Framework

This section suggest possible improvements to the evaluation framework described in chapter 6.

8.3.1 Energy Estimation

The evaluation framework currently lacks power/energy estimation capability (the thesis relied on synthesis tools for estimates). Possible starting points are Instruction-level estimation [133] and Micro-architecture-level estimation [134].

A comprehensive discussion of high-level power/energy estimation techniques is available [135].

Instruction-level estimation involves assigning a power cost to each instruction in the standard instruction set. The power cost is determined by measuring the current drawn by the CPU when an instruction is executed repeatedly in a loop. The model can be extended to account for other factors—such as the power overhead when two different instructions are sequentially executed, pipeline stalls, cache misses, etc.—by locating where the desired effect occurs in the application, running repeatedly and measuring the current drawn. However, this requires a silicon implementation of the target processor.

Micro-architecture-level estimation uses functional-block-level, activity-based, analytical models to estimate power/energy. They are often integrated into cycle-by-cycle performance simulators which supply the energy estimator with the activated micro-architecture-level units or functional blocks. Extensible micro-architecture level estimators, such as Multicore Power, Area, and Timing (McPAT) [136], can be easily integrated into the VIREMENT's evaluation framework. Accuracy could be validated with the gate-level models and power estimators.

8.3.2 Area Estimates

In addition to energy estimation, the evaluation framework could be extended to estimate area. This will speed-up architectural exploration as area estimates can be available without need for detailed gate-level models. High-level area estimators [136] often combine analytical models [137], for array structures such as caches, and empirical models[138], for more complex structures such as ALUs. A possible starting point is extending and integrating McPAT [136] into the evaluation framework.

8.4 Summary of Contributions

The major research contributions made by this thesis are:

- The identification and characterization of emerging mobile workloads and the impact on these workloads on the design of a reconfigurable accelerator.

- The VIrtual REconfigurable Micro-ENgine for Translation (VIREMENT) architecture, a mobile CMP with each core augmented with a reconfigurable accelerator. VIREMENT efficiently boosts single thread performance.
    - Architecture of the VIREMENT Reconfigurable Functional Unit (VRFU).
    - Method for the seamless integration of the VRFU into a CPU.
- Dynamic Compilation Engine (DCE) a JIT complier for VIREMENT which dynamically translates IR into configurations for the RH. This brings the advantages of reconfigurable acceleration to dynamically generated code while still meeting the stringent cost requirements of mobile computers.
    - RIG pass, a lean Placement and Routing algorithm that is suitable for resource-constrained mobile computers. This algorithm is the heart of the DCE.
    - A method for integrating the RIG into a standard dynamic complier pass pipeline.
- A standard cell ASIC implementation of the VRFU demonstrating the practicability of the design.
- An implementation of the DCE leveraging LLVM 2.8. This formed part of the evaluation framework used for gathering performance estimates.
- A demonstration of the merits of VIREMENT and DCE with a combination of simulation and analytical methods.

8.5 Conclusion

The work in this thesis has demonstrated the feasibility of using reconfigurable accelerators in resource constrained mobile processors to accelerate single-threads within dynamically generated code. The advantage of reconfigurable processing, execution efficiency, can now be extended to dynamically generated code.

VIREMENT has highlighted the difficulty of dynamic mapping on resource constrained mobile computers where the need to minimize mapping overheads reduce performance gains. Whilst this hasn't prevented VIREMENT from outperforming conventional designs, there is still the need for research on techniques for improving the mapping process.

It is hoped that based upon this thesis, reconfigurable acceleration can gain a foothold in mobile processors since it offers a clear efficiency advantage. With VIREMENT the lack of support for dynamic compilation in reconfigurable accelerators, which is a barrier to the widespread adoption of reconfigurable processing in mobile platforms, has been eliminated.

Acronyms

ALU Arithmetic and Logic Unit 29

API Application Programming Interface 93

ArBB Array Building Blocks 21

ASIC Application Specific Integrated Circuit 16

ASIP Application Specific Instruction-set Processor 16

CCA Configurable Compute Array 35

CCG Converged Code Generator 95

CMP Chip-MultiProcessor 17

CPU Central Processing Unit 17

CReAMS Custom Reconfigurable Arrays for Multiprocessor System 38

DAG Directed Acyclic Graph 78

DAP Dynamic Adaptive Processor 38

DBT Dynamic Binary Translation 22

DCE Dynamic Compilation Engine 23

DDH Dynamic Detection Hardware 38

DFG Directed Flow Graph 35

DIM Dynamic Instruction Merging 37

DIR Directly Interpretable Representation 34

EE Execution Engine 72

FPGA Field-Programmable Gate Array 26

GCC GNU Compiler Collection 95

GPGPU General Purpose Graphics Processing Unit 18

GPP General Purpose Processor 16

HDL Hardware Description Language 90

HLO High-Level Optimizer 95

HRT Heterogeneous Runtime 94

ILP Instruction Level Parallelism 19

IPC Instructions Per Cycle 49

IR Intermediate Representation 21

ISA Instruction Set Architecture 21

JIT	Just-In-Time	23
LAB	Logic Array Block	28
LE	Logical Element	28
LLO	Low-Level Optimizer	95
LLVM	Lower Level Virtual Machine	34
LLVMOOPE	LLVM Out-of-Order-Processor Emulator	68
LUT	Lookup Table	28
McPAT	Multicore Power, Area, and Timing	124
MICA	Microarchitecture-Independent Characterization of Applications	43
MIMD	Multiple Instruction Multiple Data	64
MM	Memory Manager	94
OOP	Out-of-Order-Processor	68
OS	Operating System	38
OVP	Open Virtual Platform	95
PC	Program Counter	37

PE Processing Element 27

PPM Prediction by Partial Matching 49

P&R Placement and Routing 22

RC Reconfiguration Cache 72

RFU Reconfigurable Functional Unit 32

RH Reconfigurable Hardware 21

RHU Reconfigurable Hardware Unit 38

RIG Reconfigurable Instruction Generation 38

RIG Reconfiguration Instruction Generation 38

RISC Reduced Instruction Set Computer 29

RISP Reconfigurable Instruction Set Processors 32

ROCCAD Riverside On-Chip CAD 34

RP Reconfigurable Processor 21

RTL Register Transfer Level 90

SIMD Single Instruction Multiple Data 64

SoC System-on-Chip 17

SRAM Static Random-Access Memory 28

SSA Static Single Assignment 77

TLP Thread-Level Parallelism 18

TRT Threading Runtime 94

VCU VIREMENT Control Unit 62

VEE VIREMENT Execution Environment 61

VIREMENT VIrtual REconfigurable Micro-ENgine for Translation 23

VIR VIREMENT Intermediate Representation 79

VM Virtual Machine 21

VRD VIREMENT Reconfigurable Datapath 62

VRFU VIREMENT Reconfigurable Functional Unit 60

WAR Write After Read 85

WAW Write After Write 85

References

[1] D. Weinsziehr, H. Ebert, G. Mahlich, J. Preissner, H. Sahm, J. Schuck, H. Bauer, K. Hellwig, and D. Lorenz, "KISS-16V2: a one-chip ASIC DSP solution for GSM," *IEEE Journal of Solid-State Circuits*, vol. 27, pp. 1057 – 1066, July 1992. 16

[2] "Android http://www.android.com." 16, 21, 116

[3] I. Paul, "Android market hits 450k apps, challengers abound," *PCWorld Magazine* *http://www.pcworld.com/article/250765/android_market_hits_450k_apps_challengers_abound.html*, Feb, 2012. 16

[4] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK Benchmark: past, present and future," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 9, pp. 803–820, 2003. 16

[5] J. Volpe, "Visidon Applock sees your pretty face, grants you Android access (video)," *Engadget Jun 23rd 2011*. 16

[6] T. P. Morgan, "Calxeda boasts of 5 watt ARM server Includes memory and interconnect fabric," *The Register 14th March 2011*, 2011. http://www.theregister.co.uk/2011/03/14/calxeda_arm_server. 17

[7] C. Moore, "Data processing in exascale-class computing systems (slides available at http://www.lanl.gov/orgs/hpc/salishan/salishan2011/3moore.pdf)," in *The Salishan Conference on High Speed Computing*, April 2007. 18

[8] J. L. Hennessy and D. A. Patterson, "Computer architecture - a quantitative approach (4. ed.)," 2007. 18, 97

[9] K. Olukotun and L. Hammond, "The Future of Microprocessors," *Queue*, vol. 3, pp. 26–29, Sep. 2005. 18, 20

[10] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967,*

spring joint computer conference, AFIPS '67 (Spring), (New York, USA), pp. 483–485, ACM, 1967. 19, 107, 120

[11] CRAY-1 Research Inc., *CRAY-1 Hardware Reference Manual*, 2240004 rev. c ed., Nov. 1977. 19

[12] Wikipedia, "HTC HD2 — Wikipedia, the free encyclopedia," 2012. [http://en.wikipedia.org/wiki/HTC_HD2]. 19

[13] "Top Computers Over Time for the Linpack n=100 Benchmark http://www.netlib.org/utk/people/JackDongarra/faq-linpack.html." 19

[14] "Linpack for Android http://www.greenecomputing.com/apps/linpack." 19

[15] A. Modine, "Remembering the Cray-1: When computers and furniture collide — The Register," Jan. 2008. http://www.theregister.co.uk/2008/01/05/tob_cray1. 19

[16] J. E. Thornton, "Parallel operation in the control data 6600," in *Proceedings of the October 27-29, 1964, fall joint computer conference, part II: very high speed computer systems*, AFIPS '64 (Fall, part II), (New York, USA), pp. 33–40, ACM, 1965. 19

[17] M. Wilkes, "The best way to design an automatic calculating machine," in *Manchest Univ. Computer Inaugural Conference*, pp. 16–18, July 1951. 19

[18] M. V. Wilkes and J. B. Stringer, "Micro-programming and the design of the control circuits in an electronic digital computer.," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 49, pp. 230–238, 1959. 19

[19] W. Buchholz, *Planning a computer system: Project Stretch*. Hightstown, USA: McGraw-Hill, Inc., 1962. 19

[20] G. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, April 1965. 19

[21] D. W. Wall, “Limits of Instruction-Level Parallelism,” tech. rep., Digital Western Research Laboratory, Palo Alto, CA, USA., 1993. 20

[22] N. Clark, H. Zhong, and S. Mahlke, “Processor acceleration through automated instruction set customization,” in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, (Washington, USA), pp. 129–140, IEEE Computer Society, 2003. 21, 22, 35, 36

[23] R. Lysecky, G. Stitt, and F. Vahid, “Warp processors,” in *Proceedings of the 41st Design Automation Conference*, DAC ’04, (New York, NY, USA), pp. 659–681, ACM, 2004. 21, 22, 31, 33, 34

[24] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, “Transparent reconfigurable acceleration for heterogeneous embedded applications,” in *Proceedings of the conference on Design, Automation and Test in Europe*, DATE ’08, (New York, USA), pp. 1208–1213, ACM, 2008. 21, 22, 36, 37, 49

[25] T. Suri and A. Aggarwal, “Scalable multi-cores with improved per-core performance using off-the-critical path reconfigurable hardware,” in *Proceedings of the 15th international conference on High performance computing*, HiPC’08, (Berlin, Heidelberg), pp. 365–377, Springer-Verlag, 2008. 21, 38

[26] R. E. Gonzalez, “A Software-Configurable Processor Architecture,” *IEEE Micro*, vol. 26, pp. 42–51, Sep. 2006. 21

[27] *International Technology Roadmap for Semiconductors (2001 ITRS)*. ITRS, 2001 ed., 2011. 21

[28] A. Ghuloum, A. Smith, G. Wu, X. Zhou, J. Fang, P. Guo, B. So, M. Rajagopalan, Y. Chen, and B. Chen, “Future-Proof Data Parallel Algorithms and Software on Intelfor Multi-Core Architecture,” *Intel Technology Journal*, vol. 11, pp. 333 –347, Nov. 2007. 21

[29] K. Thompson, “Programming techniques: Regular expression search algorithm,” *Commun. ACM*, vol. 11, no. 6, pp. 419–422, 1968. 22

[30] J. Aycock, “A brief history of just-in-time,” *ACM Comput. Surv.*, vol. 35, pp. 97–113, June 2003. 22

[31] *ARM Architecture Reference Manual ARMv7-A and ARMv7-R Edition*, vol. ARM DDI 0406C. 2011. 22

[32] T. Suri and A. Aggarwal, “Improving scalability and per-core performance in multi-cores through resource sharing and reconfiguration,” in *Proceedings of the 22nd International Conference on VLSI Design*, pp. 145 –150, 2009. 22, 38, 39

[33] M. B. Rutzig, A. C. S. Beck, and L. Carro, “CReAMS: an embedded multiprocessor platform,” in *Proceedings of the 7th international conference on Reconfigurable computing: architectures, tools and applications*, ARC’11, (Berlin, Heidelberg), pp. 118–124, Springer-Verlag, 2011. 22, 38

[34] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates, “FX!32: A Profile-Directed Binary Translator,” *IEEE Micro*, vol. 18, pp. 56–64, March 1998. 22, 33

[35] N. S. Voros and K. Masselos, *System Level Design of Reconfigurable Systems-on-Chip*. Secaucus, USA: Springer-Verlag New York, Inc., 2005. 26

[36] H. G. Dietz, “The Aggregate Magic Algorithms.” Aggregate.Org online technical report. 26

[37] I. Kuon and J. Rose, “Measuring the gap between FPGAs and ASICs,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, pp. 203 –215, Feb. 2007. 28

[38] *Cyclone II Device Handbook, Volume 1*. Altera Corporation., 2008. 28

[39] S. Cadambi, J. Weener, S. C. Goldstein, H. Schmit, and D. E. Thomas, “Managing pipeline-reconfigurable fpgas,” in *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, FPGA ’98, (New York, USA), pp. 55–64, ACM, 1998. 29

[40] K. Bondalapati and V. Prasanna, “Reconfigurable computing systems,” *Proceedings of the IEEE*, vol. 90, pp. 1201 – 1217, July 2002. 29

[41] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. San Francisco, USA: Morgan Kaufmann Publishers Inc., 2007. 30

[42] C. Ebeling, D. C. Cronquist, and P. Franklin, “RaPiD - Reconfigurable Pipelined Datapath,” in *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, (London, UK), pp. 126–135, Springer-Verlag, 1996. 30

[43] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, “PACT XPP—A Self-Reconfigurable Data Processing Architecture,” *J. Supercomput.*, vol. 26, pp. 167–184, September 2003. 30

[44] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh, “PRISM-II compiler and architecture,” in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, 1993*, pp. 9 –16, April 1993. 30

[45] M. J. Wirthlin, “A dynamic instruction set computer,” in *Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines*, FCCM '95, (Washington, DC, USA), pp. 99–107, IEEE Computer Society, 1995. 30

[46] S. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. Taylor, and R. Laufer, “PipeRench: a coprocessor for streaming multimedia acceleration,” in *Proceedings of the 26th International Symposium on Computer Architecture*, pp. 28 –39, 1999. 30

[47] T. Miyamori and K. Olukotun, “REMARC (abstract): reconfigurable multimedia array coprocessor,” in *Proceedings of the ACM/SIGDA 6th International Symposium on Field Programmable Gate Arrays*, FPGA '98, (New York, USA), pp. 261–, ACM, 1998. 31

[48] C. Phillips, “Wireless Base Station Design Using a Reconfigurable Communications Processor,” in *Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, FPL ’00, (London, UK), pp. 846–848, Springer-Verlag, 2000. 31

[49] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis, “The MOLEN Processor Prototype,” in *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, (Washington, DC, USA), pp. 296–299, IEEE Computer Society, 2004. 31

[50] H. Singh, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Filho, “MorphoSys: a reconfigurable architecture for multimedia applications,” in *Proceedings of the XI Brazilian Symposium on Integrated Circuit Design*, pp. 134 –139, 1998. 31, 32, 52

[51] R. Wittig and P. Chow, “Onechip: an FPGA processor with reconfigurable logic,” in *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines*, FCCM ’96, pp. 126 –135, April 1996. 32, 90

[52] S. Hauck, T. Fry, M. Hosler, and J. Kao, “The chimaera reconfigurable functional unit,” in *Proceedings of the 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 87 –96, April 1997. 32, 90

[53] F. Barat and R. Lauwereins, “Reconfigurable instruction set processors: a survey,” in *Proceedings of the 11th International Workshop on Rapid System Prototyping*, RSP 2000, pp. 168 –173, 2000. 32

[54] A. Olugbon, T. Arslan, I. Lindsay, and S. MacDougall, “Providing compilers and application program support for reconfigurable SoCs: Radical but overdue,” in *Proceedings of the 2005 International Symposium on System-on-Chip*, pp. 54 –57, Nov. 2005. 33

[55] F. Vahid, G. Stitt, and R. Lysecky, “Warp processing: Dynamic translation of binaries to FPGA circuits,” *Computer*, vol. 41, pp. 40 –46, July 2008. 33, 34

[56] C. Cifuentes and K. J. Gough, "Decompilation of binary programs," *Softw. Pract. Exper.*, vol. 25, pp. 811–829, July 1995. 34

[57] G. Stitt and F. Vahid, "Binary synthesis," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, pp. 34:1–34:30, May 2008. 34

[58] B. R. Rau, "Levels of representation of programs and the architecture of universal host machines," in *Proceedings of the 11th Workshop on Microprogramming*, MICRO 11, (Piscataway, USA), pp. 67–79, IEEE Press, 1978. 34

[59] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '04, (Washington, USA), pp. 75–86, IEEE Computer Society, 2004. 34, 58, 76

[60] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, "Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, (Washington, DC, USA), pp. 30–40, IEEE Computer Society, 2004. 35, 36

[61] E. Rotenberg, S. Bennett, and J. Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching," in *Proceedings of the 29thIEEE/ACM International Symposium on Microarchitecture*, MICRO 29, pp. 24 –34, December 1996. 36

[62] S. J. Patel and S. S. Lumetta, "rePLay: A hardware framework for dynamic optimization," *IEEE Trans. Comput.*, vol. 50, pp. 590–608, June 2001. 36

[63] B. Slechta, D. Crowe, N. Fahs, M. Fertig, G. Muthler, J. Quek, F. Spadini, S. Patel, and S. Lumetta, "Dynamic optimization of micro-operations," in *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture*, HPCA-9, pp. 165 – 176, feb. 2003. 38

[64] S. Hu, I. Kim, M. Lipasti, and J. Smith, "An approach for implementing efficient superscalar cisc processors," in *International Symposium on High-*

Performance Computer Architecture, vol. 0, (Los Alamitos, USA), pp. 41–52, IEEE Computer Society, 2006. 38

[65] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, (New York, USA), pp. 72–81, ACM, 2008. 40, 43

[66] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, IISWC '09, (Washington, USA), pp. 44–54, IEEE Computer Society, 2009. 40

[67] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP," in *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, (Washington, USA), pp. 124–131, IEEE Computer Society, 2009. 40, 41

[68] "Intel Threading Building Blocks http://threadingbuildingblocks.org." 41

[69] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Parallel distributed processing: explorations in the microstructure of cognition, vol. 1*, ch. Learning internal representations by error propagation, pp. 318–362. 1986. 40

[70] E. Sifakis, I. Neverov, and R. Fedkiw, "Automatic determination of facial muscle activations from sparse motion capture marker data," pp. 417–425, 2005. 41

[71] J. Cooley and J. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of Computation*, vol. 19, pp. 297–301, April 1965. 41

[72] M. Müller, D. Charypar, and M. Gross, "Particle-based fluid simulation for interactive applications," in *Proceedings of the 2003 ACM SIG-*

GRAPH/Eurographics Symposium on Computer Animation, SCA ’03, (Aire-la-Ville, Switzerland), pp. 154–159, Eurographics Association, 2003. 42

[73] G. Grahne and J. Zhu, “Frequent itemset mining implementations,” in *Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations* (B. Goethals and M. J. Zaki, eds.), vol. 90 of *FIMI ’03*, (Melbourne , USA), CEUR-WS, December 2003. 42

[74] F. Stahl, M. M. Gaber, M. Bramer, and P. S. Yu, “Pocket data mining: Towards collaborative data mining in mobile computing environments,” in *Proceedings of the 22nd IEEE International Conference on Tools with Artificial Intelligence - Volume 02*, vol. 2 of *ICTAI ’10*, (Washington, DC, USA), pp. 323–330, IEEE Computer Society, 2010. 42

[75] M. Jang, M.-S. Han, J.-h. Kim, and H.-S. Yang, *Dynamic Time Warping-Based K-Means Clustering for Accelerometer-Based Handwriting Recognition*. Studies in Computational Intelligence, Springer Berlin/Heidelberg, 2011. 42

[76] S. G. Akl and N. Santoro, “Optimal parallel merging and sorting without memory conflicts,” *IEEE Trans. Comput.*, vol. 36, pp. 1367–1369, Nov. 1987. 42

[77] Y. Yu and S. Acton, “Speckle reducing anisotropic diffusion,” *IEEE Transactions on Image Processing*, vol. 11, pp. 1260 – 1270, Nov. 2002. 42

[78] “Mobisante Inc `http://www.mobisante.com`.” 42

[79] K. Martinez and J. Cupitt, “VIPS - a highly tuned image processing software architecture,” Sept. 2005. 43

[80] “VideoLAN organisation `http://www.videolan.org/developers/x264.html`.” 43

[81] “Intel ™ software development emulator. `http://software.intel.com/en-us/articles/intel-software-development-emulator`.” 43

[82] K. Hoste and L. Eeckhout, “Microarchitecture-independent workload characterization,” *Micro, IEEE*, vol. 27, pp. 63 –72, may-june 2007. 43, 49, 52

[83] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: building customized program analysis tools with dynamic instrumentation,” in *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI ’05, (New York, NY, USA), pp. 190–200, ACM, 2005. 43

[84] I.-C. K. Chen, J. T. Coffey, and T. N. Mudge, “Analysis of branch prediction via data compression,” in *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ASPLOS-VII, (New York, USA), pp. 128–137, ACM, 1996. 49

[85] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg, “Data and memory optimization techniques for embedded systems,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 6, no. 2, pp. 149–206, 2001. 53

[86] K. Keutzer, S. Malik, and A. R. Newton, “From ASIC to ASIP: The Next Design Discontinuity,” in *Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, ICCD ’02, (Washington, USA), pp. 84–, IEEE Computer Society, 2002. 58

[87] M. Milkes, “The genesis of microprogramming,” *Annals of the History of Computing*, vol. 8, pp. 116 –126, april-june 1986. 58

[88] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, “The superblock: an effective technique for VLIW and superscalar compilation,” vol. 7, (Hingham, USA), pp. 229–248, Kluwer Academic Publishers, May 1993. 60, 80, 123

[89] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. 30, pp. 478–490, July 1981. 60, 80, 123

[90] *ARM926EJ-S Technical Reference Manual*, vol. ARM DDI 0198E. revision: r0p5 ed., June 2008. 60

[91] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," in *Proceedings of the 11th International Symposium on Computer Architecture*, ISCA '84, (New York, NY, USA), pp. 348–354, ACM, 1984. 61

[92] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, pp. 66–76, December 1996. 61

[93] M. Ahn, J. W. Yoon, Y. Paek, Y. Kim, M. Kiemb, and K. Choi, "A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures," in *Proceedings of the conference on Design, Automation and Test in Europe: Proceedings*, DATE '06, (Leuven, Belgium), pp. 363–368, European Design and Automation Association, 2006. 63

[94] C. O. Shields, Jr., *Area efficient layouts of binary trees in grids*. PhD thesis, 2001. AAI3015147. 63

[95] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in *Proceedings of the conference on Design, Automation and Test in Europe*, DATE '01, (Piscataway, NJ, USA), pp. 642–649, IEEE Press, 2001. 64

[96] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," in *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, (Washington, DC, USA), pp. 10296–, IEEE Computer Society, 2003. 64

[97] Y. Kim, J. Lee, A. Shrivastava, and Y. Paek, "Operation and data mapping for cgras with multi-bank memory," in *Proceedings of the ACM SIG-*

PLAN/SIGBED 2010 conference on Languages, Compilers, and Tools for Embedded Systems, LCTES '10, (New York,USA), pp. 17–26, ACM, 2010. 71

[98] T. Juan, J. J. Navarro, and O. Temam, "Data caches for superscalar processors," in *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, (New York, USA), pp. 60–67, ACM, 1997. 71

[99] A. Cohen and E. Rohou, "Processor virtualization and split compilation for heterogeneous multicore embedded systems," in *Proceedings of the 47th Design Automation Conference*, DAC '10, (New York, USA), pp. 102–107, ACM, 2010. 77, 122

[100] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, pp. 451–490, Oct. 1991. 77

[101] *AMD Accelerated Parallel Processing OpenCL*. Sunnyvale, CA, USA.: Advanced Micro Devices, Inc., August 2011. 77

[102] "The LLVM Target-Independent Code Generator `http://llvm.org/docs/CodeGenerator.html`." 77, 78, 79

[103] M. Lam, "Software pipelining: an effective scheduling technique for vliw machines," in *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language Design and Implementation*, PLDI '88, (New York, NY, USA), pp. 318–328, ACM, 1988. 80, 123

[104] "gem5 `http://www.m5sim.org`." 90, 91, 92

[105] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011. 90, 91, 92

[106] M. Bocchi, C. De Bartolomeis, C. Mucci, F. Campi, A. Lodi, M. Toma, R. Canegallo, and R. Guerrieri, “A XiRisc-based SoC for embedded DSP applications,” in *Proceedings of the IEEE Custom Integrated Circuits Conference*, pp. 595 – 598, Oct. 2004. 90

[107] “IEC/IEEE behavioural languages-part 4: Verilog hardware description language (adoption of IEEE Std 1364-2001),” *IEC 61691-4 First edition 2004-10; IEEE 1364*, pp. 1–860, 2004. 91, 92

[108] “Synopsys Inc. `http://www.synopsys.com`.” 91, 92, 99

[109] M.-H. Wu, P.-C. Wang, C.-Y. Fu, and R.-S. Tsay, “An extended systemc framework for efficient hw/sw co-simulation,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 17, pp. 11:1–11:16, April 2012. 91

[110] “Si2 `http://www.si2.org`.” 92, 99

[111] “The SimpleScalar-Arm Power Modeling Project `http://web.eecs.umich.edu/~panalyzer`.” 92

[112] B. Bailey, M. Grant, and T. Anderson, eds., *Taxonomies for the development and verification of digital systems*. New York, USA: Springer, 2005. 93

[113] B. Bailey, “White Paper: System Level Virtual Prototyping & OVP,” tech. rep., 2008. 93

[114] C. J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. D. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang, “Intel’s array building blocks: A retargetable, dynamic compiler and embedded language,” in *Proceedings of the 29th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO ’11, (Washington, DC, USA), pp. 224–235, IEEE Computer Society, 2011. 93

[115] “Building the LLVM GCC Front-End `http://llvm.org/docs/GCCFEBuildInstrs.html`.” 95

[116] “The GNU Complier Collection `http://gcc.gnu.org`.” 95

[117] "Open Virtual Platform http://www.ovpworld.org." 96

[118] K. Smart, "The life cycle of a virtual platform," in *Processor and System-on-Chip Simulation* (R. Leupers and O. Temam, eds.), ch. 3, pp. 25–45, Springer US, 2010. 96

[119] F. Koushanfar, V. Prabhu, M. Potkonjak, and J. Rabaey, "Processors for mobile applications," in *Proceedings of the International Conference on Computer Design*, ICCD' 00, pp. 603 –608, Sept. 2000. 97

[120] F. N. Najm, "Transition density, a stochastic measure of activity in digital circuits," in *Proceedings of the 28th ACM/IEEE Design Automation Conference*, DAC '91, (New York, USA), pp. 644–649, ACM, 1991. 99

[121] I. Nedelchev and B. Chen, "Power compiler: a gate-level power optimization and synthesis system," in *Proceedings of the 1997 International Conference on Computer Design*, ICCD '97, (Washington, DC, USA), pp. 74–, IEEE Computer Society, 1997. 99

[122] D. Brand and C. Visweswariah, "Inaccuracies in power estimation during logic synthesis," in *Proceedings of the 1996 IEEE/ACM international conference on Computer-Aided design*, ICCAD '96, (Washington, DC, USA), pp. 388–394, IEEE Computer Society, 1996. 99

[123] A. T. Schwarzbacher, P. A. Comiskey, and J. B. Foley, "Powercount : measuring the power at the VHDL netlist level," in *Electronic Devices and Systems Conference*, 1998. 99

[124] *Power Compiler User Guide version F-2011.09-SP4*. Synopsys Inc, 2012. 99

[125] J. Flynn, "Accurate power-analysis techniques support smart SoC-design choices," *EDN Magazine*, Dec. 2007. 99

[126] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of the 44th annual Design Automation Conference*, DAC '07, (New York, USA), pp. 746–749, ACM, 2007. 103

[127] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai, “Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus?,” in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO ’43, (Washington, DC, USA), pp. 225–236, IEEE Computer Society, 2010. 106

[128] E. Grochowski and M. Annavaram, “Energy per instruction trends in intel™ microprocessors.,” *Technology@Intel Magazine*, March 2006. 106

[129] D. H. Woo and H.-H. S. Lee, “Extending Amdahl’s Law for Energy-Efficient Computing in the Many-Core Era,” *Computer*, vol. 41, pp. 24–31, Dec. 2008. 107, 108, 109, 110, 111

[130] Microsoft Inc, *Application Certification Requirements for Windows Phone* http://msdn.microsoft.com/en-us/library/hh184843(v=vs.92), May 2012. 116

[131] D. Lee, M. Jo, K. Han, and K. Choi, “FloRA: Coarse-grained reconfigurable architecture with floating-point operation capability,” in *Proceedings of the 2009. International Conference on Field-Programmable Technology*, FPT ’09, pp. 376 –379, Dec. 2009. 122

[132] P. Lesnicki, A. Cohen, and G. Fursin, “Split Compilation: an Application to Just-in-Time Vectorization Abstract,” in *International Workshop on GCC for Research in Embedded and Parallel System*, Sept. 2007. 122

[133] V. Tiwari, S. Malik, and A. Wolfe, “Power analysis of embedded software: a first step towards software power minimization,” in *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, ICCAD ’94, (Los Alamitos, USA), pp. 384–390, IEEE Computer Society Press, 1994. 123

[134] D. Brooks, V. Tiwari, and M. Martonosi, “Wattch: a framework for architectural-level power analysis and optimizations,” in *Proceedings of the 27th International Symposium on Computer Architecture*, ISCA ’00, pp. 83 –94, June 2000. 123

[135] P. Landman, “High-level power estimation,” in *Proceedings of the 1996 international symposium on Low Power Electronics and Design*, ISLPED ’96, (Piscataway, USA), pp. 29–35, IEEE Press, 1996. 124

[136] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, “McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures,” in *Proceedings of the 42nd IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, (New York, USA), pp. 469–480, ACM, 2009. 124

[137] P. Shivakumar, N. P. Jouppi, and P. Shivakumar, “CACTI 3.0: An Integrated Cache Timing, Power, and Area Model,” tech. rep., Western Research Laboratory, 2001. 124

[138] S. Gupta, S. W. Keckler, and D. Burger, “Technology independent area and delay estimates for microprocessor building blocks,” tech. rep., University of Texas Austin, 2000. 124