RUN-TIME OBJECT CODE COMPILATION TO HARDWARE

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2008

By
Ian Jason
School of Computer Science

Contents

List of Figures

List of Tables

Abstract

Reconfigurable hardware systems offer great potential for improving performance over software through increased parallelism and reduced control overhead. However, the configuration of the hardware is performed statically, restricting the system's ability to adapt to the run-time environment. This thesis explores the possibility of compiling from object code into a reconfigurable hardware structure at run-time to allow the hardware configuration to benefit from the behaviour of the currently executing code.

A mechanism for detecting hot spots at run-time is presented; this minimises compilation overheads by only compiling the most frequently executed sections of code. A dynamic detection threshold allows the hot spot detector to adapt to radically different code and to react quickly to changes in program behaviour. A hardware structure suitable for executing these hot spots is proposed, consisting of a sequenced execution engine containing four parallel functional units and a configurable interconnect.

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 institution of learning.

Copyright

Acknowledgements

Firstly I would like to thank my supervisor Dr. Jim Garside for his advice, insight and patience over the last four years.

Thanks must also go to all my friends in the APT group for keeping me sane, especially Sam (for his inspirational research methodology), Andrew and Matt.

Special thanks are due to my parents for their encouragement, support and tolerance.

Finally, I would like to thank Sonja for inspiring me to finish this thesis.

Chapter 1

Introduction

Today's consumers demand an increasingly diverse set of features in portable electronic devices. Modern mobile phones, for example, are capable not only of making and receiving calls but also playing music and video, taking photographs and browsing the Internet. These features require considerable computing resources; in the past the significant data processing tasks performed in such devices, such as voice processing, were executed in dedicated hardware processors. Dedicated hardware can perform a task much more quickly and efficiently than a microprocessor executing software by both increasing parallelism and by reducing the control overhead required by executing an algorithm in software.

Developing custom hardware has a number of disadvantages however: systems are expensive and time-consuming to develop, and once such a system is produced it is not possible to modify it if required, for example in the event of a defect being discovered or additional features being required. The increasing diversity in the processing requirements of portable devices means that it is not feasible to develop custom hardware for them all; instead a more general purpose processing unit is required.

Reconfigurable hardware provides performance similar to fixed-function hardware while maintaining flexibility close to that of a microprocessor. Initial reconfigurable systems were based on standalone Field Programmable Gate Arrays (FPGAs) intended as a soft-programmable replacement for fixed-function hardware; more recent systems are hybrid architectures, consisting of a microprocessor with a reconfigurable coprocessor. These allow the core algorithms to be accelerated by the reconfigurable hardware and the less frequently executed parts of a program to execute in software, reducing the amount of reconfigurable hardware required.

Selecting the sections of code to compile into the reconfigurable hardware is typically performed at compile-time using static profile information. However, a program's behaviour is not always easily predictable at compile-time: it may depend, for example, on the data being processed or options selected by the user. Additionally, during execution a program's behaviour may not be constant as it may consist of a number of phases when different sections of code are executed.

Backward compatibility is an additional disadvantage of reconfigurable architectures as special programming methods and tools are required to create configurations for the reconfigurable hardware. Incompatibility with the existing code base provides a significant barrier to adoption, as evidenced by the continued prevalence of Intel's x86 instruction set architecture despite the development of more efficient alternatives.

One possible solution to these problems is to translate the code at run-time. This is used in dynamic binary translation systems to allow legacy code to execute on new architectures. Similar techniques are used in dynamic optimisation systems, which transform instructions at run-time into a more efficient internal representation to improve performance.

However, performing the optimisation at run-time is an expensive process and may incur significant overheads if performed for an entire program. These overheads can be reduced by only selecting the most frequently executed parts of a program for optimisation. This can be achieved by using *dynamic* *profiling* mechanisms; these monitor the executing program to locate the most frequently executed sections of code, or *hot spots*, which are then optimised.

1.1 Research goals

This thesis explores the feasibility and potential benefits of compiling existing object code into reconfigurable hardware at run-time. By profiling the currently executing program, hot spots are detected which are then compiled into a reconfigurable execution engine. Key contributions include:

- The development of architectural techniques to allow object code to be compiled at run-time into reconfigurable hardware.

- Analysis of program execution, leading to a proposed mechanism for detecting program hot spots at run-time, including a method of dynamically altering the threshold of the hot spot detector to quickly adapt to variations in program behaviour.

- The use of information gathered dynamically by the hot spot detector to detect highly biased branches and to use this to speculatively combine multiple basic blocks into a single atomic entity with a single entry and exit point. An analysis of optimisations that can be performed within these blocks is given, and mechanisms for handling incorrect speculation are proposed.

- The development of a reconfigurable parallel execution engine capable of processing detected hot spots. Methods for overcoming difficulties of executing hardware configurations created from object code, such as supplying immediate operands and detecting memory aliasing, are explored.

- An evaluation of the potential performance improvements that can be gained by dynamic compilation into hardware using the techniques described in this thesis.

1.2 Simulator and benchmark details

The system described in this thesis is evaluated by modifying a software model of an ARM [JS00] microprocessor. The ARM architecture is chosen as it is commonly used in real-world embedded systems that the techniques in this thesis are intended for, and tools (such as simulators and compilers) are widely available. The simulator models the architecture at the instruction level: detailed simulation of low-level architectural structures such as pipeline stage interaction is not performed.

Four benchmarks are used to evaluate the system[1]. These were selected to give a varied set of different program behaviours:

Blowfish is an encryption algorithm [Sch93]. The input data is a 100KB file, causing the benchmark to execute 15 million instructions. *Blowfish* contains a very small set of basic blocks which make up the majority of the dynamic trace of the program; this property allows it to perform well in this system.

Cjpeg is an open-source implementation of the jpeg image compression standard [cjp]. This benchmark runs for 134 million instructions when compressing a 1.4MB TGA file using default settings. Like *blowfish* this benchmark contains a number of very frequently executed hot spots; however, unlike *blowfish* distinct phases of operation are present, causing a change in the working set of basic blocks about a third of the way through the program's execution.

[1]Contact author at ianjason(at)gmx.at to request input data used with benchmarks.

Nasm is an x86 assembler [nas]. 14 million ARM instructions are required to assemble the 500 line program used as test data. This benchmark is provided as a contrast to the other benchmarks as it performs a larger, diverse set of operations on a relatively small input. This program contains no hot spots that dominate program execution time and so performs poorly in this system compared to the other benchmarks.

Qsort is a program to sort words in an input text file into alphabetical order. The input data to be sorted contains 13000 words, and the benchmark runs for 25 million instructions. Like *cjpeg*, *qsort* contains distinct phases of execution, but program behaviour fluctuates significantly within these phases.

All benchmarks are compiled with GCC 3.4.3 [gcc] using level two optimisations.

1.3 Thesis outline

The next chapter outlines developments in reconfigurable computing and dynamic optimisation systems. Chapter 3 describes the process of monitoring code and detecting hot spots. Optimisations that can be performed on detected hot spots are discussed in chapter 4. A suitable reconfigurable execution engine for processing these hot spots is outlined in chapter 5. Chapter 6 combines the techniques described in the preceding chapters to analyse the performance of such a system. Finally, chapter 7 draws conclusions on the work presented in this thesis, identifies limitations and outlines directions for future work.

Chapter 2

Reconfigurable hardware and dynamic compilation

This chapter presents an overview of reconfigurable hardware and dynamic compilation techniques. The concept of reconfigurable hardware is outlined and compared with fixed-function hardware and general-purpose microprocessors. A description of different types of reconfigurable hardware and methods of use is given. A number of examples of architectures utilising reconfigurable hardware are described briefly.

The second part of this chapter deals with dynamic compilation and optimisation techniques; their use in translation for binary compatibility and dynamic optimisation to enhance performance is described. A number of example systems are outlined, both hardware and software based.

2.1 Reconfigurable computing

In traditional computing systems two primary methods are used to execute algorithms. One is to encode the algorithm into a series of software instructions which are used to control a microprocessor. Microprocessors are very flexible: changing the algorithm to be processed merely involves changing the software. However, this limits performance as the algorithm is executed sequentially, and decoding the stream of instructions incurs overheads due to extra instructions other than those used to process data being required to load data into the hardware and control the execution of the algorithm.

These overheads can be removed by creating a piece of hardware specifically tailored to process the algorithm required. The algorithm can be processed in parallel as the requirement for it to be encoded as a sequence of instructions is removed. No instructions must be fetched and decoded as the function of the algorithm is encoded into the structure of the hardware itself. Executing an algorithm in custom hardware can therefore be significantly faster and more power efficient than the same algorithm executing in a microprocessor. Custom hardware is therefore used when high performance or low power consumption is required and the algorithm is inherently parallel; for example processing video, routing network packets or processing voice data in a mobile phone. However, a specific custom-designed piece of hardware can only process one algorithm; should a different algorithm be required a new hardware design must be produced at significant cost.

Reconfigurable hardware offers a compromise between these two methods. The structure of a reconfigurable hardware system is not fixed: it can be soft-configured to process different algorithms. Once configured, reconfigurable hardware can be used to process an algorithm in a similar way to custom hardware, with no need to continually fetch and decode instructions. Data can pass directly between execution units, and any inherent parallelism in the algorithm can be more easily exploited as there is no need to follow a linear sequence of instructions.

These factors should allow reconfigurable hardware to achieve significantly higher performance than a microprocessor while maintaining greater flexibility than hardware.

2.1.1 Field programmable gate arrays

Field programmable gate arrays (FPGAs) [BR96] are an example of a reconfigurable hardware structure. FPGAs consist of a grid of *logic blocks* and a two-dimensional programmable interconnect. Logic blocks typically consist of look-up tables (LUTs), which can produce any function of a number of inputs allowing general logic functions to be computed. Fixed-function logic blocks containing commonly used structures, such as adders, can be included to improve performance at the cost of some flexibility. The interconnect provides much of the flexibility of an FPGA as it consists of a hierarchical grid of horizontal and vertical connections allowing signals to be routed almost anywhere in the array.

FPGAs were originally intended for low product volumes where producing custom hardware would be uneconomical, or for rapidly prototyping hardware designs. However, it has been shown that FPGAs can be used as an alternative computing paradigm and have been demonstrated to show significantly improved performance in some applications traditionally employing software [Gos96][EP00][GHK$^+$91].

Reconfigurable hardware is not limited to FPGAs however. Different logic block sizes and interconnect structures can be used to tailor the reconfigurable fabric for the intended application. Many systems have demonstrated the effectiveness of combining reconfigurable structures with a traditional microprocessor (see section 2.1.3).

2.1.2 Granularity

The *granularity* of a reconfigurable hardware structure refers to the size of configurable logic blocks in the structure. A *fine-grained* structure contains small logic blocks that are configurable at the bit-level, usually consisting of small look-up tables. These have a very high flexibility and are suited to

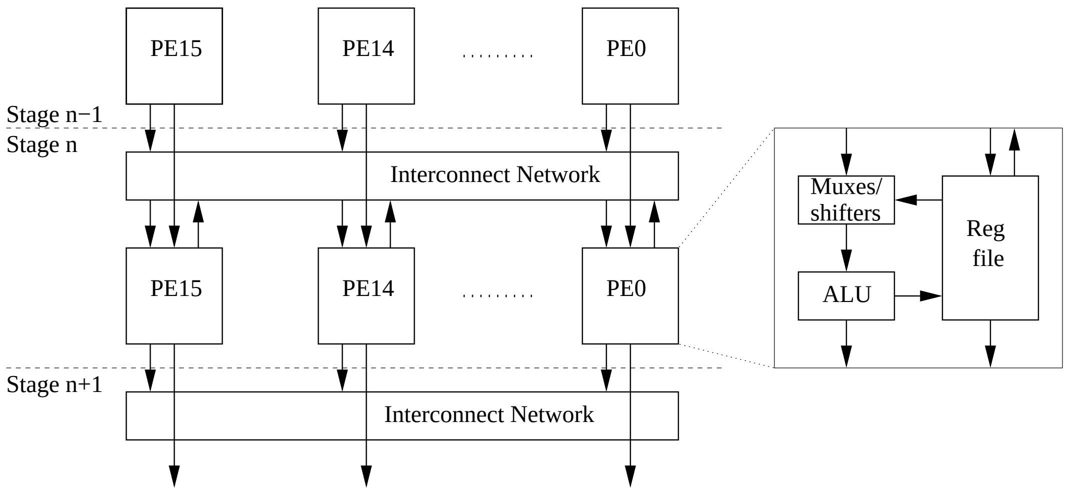


Figure 2.1: PipeRench reconfigurable datapath [GSM+99]

algorithms that manipulate individual bits, such as encryption. The hardware can also be easily tailored to match the width of the data being processed. In addition, fine granularity is required for random logic and state machines, hence the use of fine-grained reconfigurable structures such as FPGAs for prototyping hardware designs.

Coarse-grained reconfigurable hardware contains larger, fixed-function logic blocks such as adders, shifters and multipliers [Har01]. These are less flexible than fine-grained architectures as individual bits cannot be configured. However, a large fixed function logic block is faster and more power efficient than a series of interconnected look-up tables and so performance is greater when executing algorithms that process fixed word-size data using the functions available in the logic blocks. The Pleiades architecture [ZPG+00] is very coarse-grained, consisting of a number of heterogeneous processing units optimised for different tasks in a reconfigurable interconnect structure. PipeRench [GSM+99] is a coarse-grained reconfigurable pipeline architecture; each pipeline stage consists of sixteen configurable 8-bit ALUs, or Processing Elements (PEs), and a reconfigurable interconnect (see figure 2.1).

The amount of configuration data in a coarse-grained structure is less than an equivalent fine-grained structure. This is due to commonality in the operations performed, requiring the configuration of fewer, larger logic blocks. This commonality is extended to the interconnect, which routes data in bundles as opposed to individual bits.

The granularity of a reconfigurable hardware structure is therefore a trade-off between performance, flexibility and configuration effort. The reconfigurable architecture proposed in chapter 5 of this thesis is coarse grained, consisting of word-width functional units within a reconfigurable interconnect.

2.1.3 Coupling with host processor

Reconfigurable hardware can stand alone without a microprocessor; this is usually the case if the reconfigurable hardware is replacing a custom hardware unit in a system. Many algorithms that are designed to run in software cannot be mapped entirely into a block of reconfigurable hardware due to limited space. A solution to this is to map the frequently executed core parts of an algorithm into the reconfigurable hardware and have the remaining infrequently executed code execute in software.

Reconfigurable coprocessors

One way to achieve this is to have a microprocessor with a reconfigurable coprocessor. This provides a trade off between accelerating the performance of a program and keeping the size of the required reconfigurable fabric low. Garp [HW97] is an example of this type of architecture. Garp incorporates an FPGA-like reconfigurable structure with a MIPS processor. The reconfigurable hardware is loaded with configurations for executing the core algorithms of an application and acts as a coprocessor to the MIPS core, which executes the remaining parts of the code. Configuration of the Garp array is done statically:

configurations are created at compile time and then are loaded at run-time under microprocessor control. Only one configuration can be active at any one time; rapid switching between configurations is facilitated by caching multiple sets of configuration data. Many other examples of reconfigurable coprocessor architectures exist [Pag94][MO99][RLG$^+$98][SLL$^+$00].

Reconfigurable functional units

Another method is to use reconfigurable functional units within the microprocessor core itself [RS94][HFHK97][WH95][WC96]. Reconfigurable functional units are either used to provide custom complex instructions to accelerate a frequent sequence of operations within an application, or used to alter the number of parallel functional units of a certain type within a superscalar processor to match the distribution of operations in the executing program [NZ04].

2.1.4 Configuring the hardware

A significant drawback of FPGAs is that configuration data are only compatible with a specific device. This means configurations created for one generation of FPGAs will need to be recreated to work on the next generation of larger, faster devices. A number of architectures have attempted to solve this problem by virtualising the physical hardware.

PipeRench [GSM$^+$99] divides a configuration up into virtual pipeline stages; these are then mapped into a physical reconfigurable hardware pipeline. If insufficient physical stages are available multiple configurations can be swapped in and out, effectively allowing an unlimited number of virtual stages. Performance is improved by having a greater number of physical stages, allowing it to scale with improvements in technology.

SCORE [CCH$^+$00] uses a similar method of virtualisation. A hardware representation of an algorithm is divided up into virtual *compute pages* which

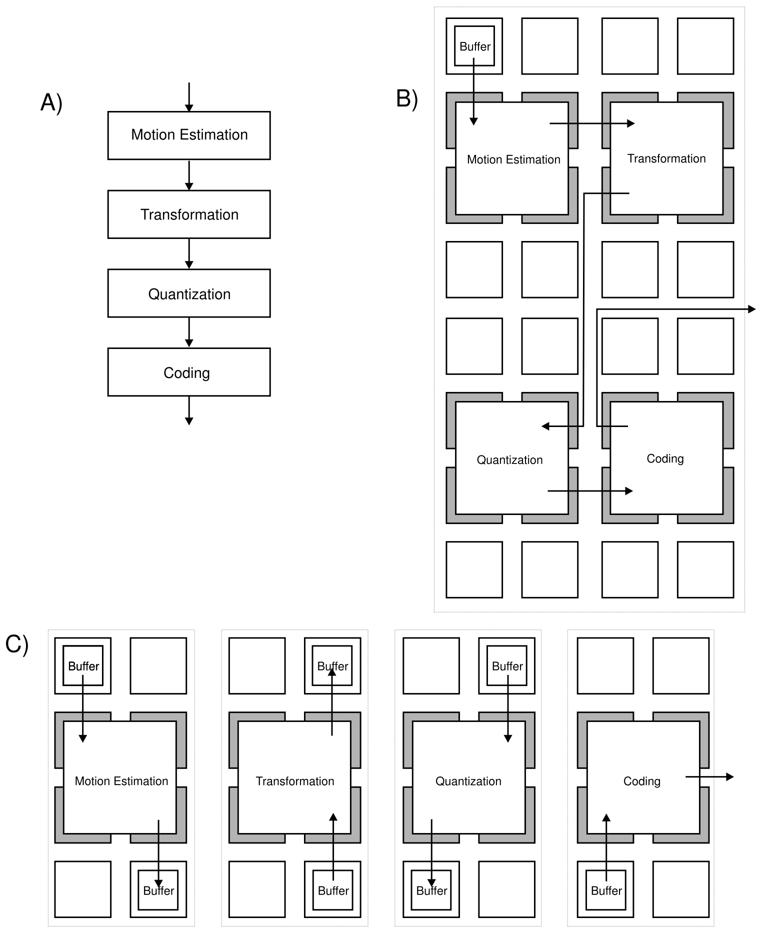


Figure 2.2: SCORE reconfigurable hardware virtualisation. A) Stages in video processing algorithm. B) Compute pages loaded simultaneously into large hardware structure. C) Compute pages time-multiplexed into smaller hardware structure. [CCH$^+$00]

interface in a data flow manner. These are mapped onto physical FPGA-like compute pages; if there are insufficient physical pages they can be time-multiplexed by swapping configurations in and out of the hardware (see figure 2.2). Data is buffered in on-chip memories between compute pages.

SCORE and PipeRench are both heavily pipelined and need to be supplied with a constant stream of data to perform well. They are therefore targeted towards algorithms that access data in predictable patterns such as those found in streaming media applications and DSPs, limiting their use. They also require special programming methods and tools to create the hardware configurations.

This thesis proposes a reconfigurable architecture configured by translating native code at run-time into hardware configurations; these are then stored on-chip and loaded on demand into the reconfigurable hardware.

2.1.5 Hardware/software partitioning

In systems that contain both a microprocessor and a reconfigurable structure the application must be partitioned into software and hardware sections. Selecting the procedures or loops to be executed in hardware is usually performed manually by the programmer. Static profiling can be used to help select the appropriate sections of code. However, this requires program behaviour at run-time to match that at the time of simulation, which may not be the case with different input data or user options.

Creating the hardware configurations typically requires special design processes and tools, and requires a different programming model for the hardware and software sections of the program. The executable is divided into two parts: one contains the software instructions that execute on the microprocessor and the other contains configuration data for the hardware. These data are loaded into the reconfigurable hardware by special instructions in the software.

HASTE [Lev05] differs by using a single programming model and executable for both the sequential processor and the reconfigurable hardware. The instruction set architecture is designed to be easily translatable to reconfigurable hardware while remaining efficiently executable in a microprocessor. During execution, core loops of a program are translated into the reconfigurable fabric by a dedicated hardware unit while the first iteration of the loop is executed on the sequential processor. The remaining iterations of the loop are then executed in the reconfigurable hardware. However, these core loops must be selected statically at compile time.

Reconfigurable systems have been shown to achieve significant benefits over traditional microprocessors [CH02]. The core algorithms that are compiled into the hardware are selected manually or using static profiling techniques; if program behaviour differs at run-time from during profiling, for example if it is dependent on input data, then the compiled hot spots may not match the hot spots executed at run-time. Additionally, different programming models are required to create hardware configurations and so these systems are not compatible with existing object code or tool-chains.

The system proposed in this thesis attempts to use reconfigurable hardware to improve performance by using *dynamic optimisation* to translate frequently executed sections of the code into a reconfigurable hardware structure at run-time. This requires additional hardware to detect hot spots at run-time, but means that only sections of code which are frequently executed during the current invocation of the program will be translated. This also allows the use of existing compiled code, removing the requirement for new programming methods and tools and maintaining backwards compatibility with legacy code.

2.2 Dynamic optimisation

Dynamic optimisation is the process of monitoring code behaviour as it executes and then using this information to perform optimisations on the executing program. Optimising at run-time can bring a number of benefits over performing optimisations at compile-time. Profile information can be used to target optimisations at regions of code that can gain the greatest improvement. This can include optimisations that may only give benefits in certain circumstances, such as loop unrolling and data prefetching.

Profiling and optimising at run-time incurs an overhead; this must be overcome by the optimisations performed. The profile information can be used to direct optimisations towards code sections that execute most frequently; these are known as *hot spots*. Code sections that execute infrequently can be left unoptimised, thus reducing the overhead of the optimisation process.

Dynamic optimisation can also be used in conjunction with a binary translator to allow executables compiled for one instruction set to be executed transparently on an incompatible microprocessor. Optimisations are performed on translated sequences of instructions to ameliorate the performance overheads of translation. Dynamic translation is typically used to allow incompatible legacy code to run on current microprocessors; however, it can also be used to improve performance of existing binaries by allowing them to execute on a more efficient microprocessor architecture [DGB$^+$03][EA97].

2.2.1 Dynamic optimisation in software

Dynamic optimisation may be performed by software, hardware or a combination of the two. Dynamo [BDB00] is an example of a software-only dynamic optimisation system. Initially code is interpreted and likely hot spot start points such as backward branch targets (which are likely to be loop entry points) are monitored. Once a monitored point's execution count exceeds a threshold value, a trace is recorded as the instructions executed after this point are interpreted. Trace construction ends when a backward branch or a branch into an already constructed trace is encountered. Optimisations that can not be, or are not usually, performed statically, such as loop unrolling and function inlining from dynamically linked libraries, are then performed on the trace. Optimised fragments of code are then stored in a software fragment cache and executed natively on the processor (instead of being interpreted) the next time they are reached.

Dynamo operates on native code: no translation is performed. However, similar techniques are used in dynamic translation systems. Some interpret and monitor code until hot spots are detected and only translate frequently executed sections to native instructions [CH97]; others, such as DAISY [EA97] and Transmeta's Crusoe [Kla00] architecture translate code as it is encountered and cache translated traces for future re-use [HKZ+06][DGB+03][EA97]. Dynamic optimisation is also used to improve the performance of Java virtual machines [BH03][AFG+00].

These systems must either interpret the code in software or translate all of the code to native instructions: this impacts on performance as interpretation is slow, and translating is a time consuming process which, if performed on large sections of code that execute infrequently, will have a large performance overhead. The dynamic optimisation system proposed in this thesis differs from these software-based dynamic optimisation systems by adding hardware to perform profiling. This allows the unoptimised code to execute in the microprocessor as opposed to being interpreted while still providing profiling support, so that only frequently executed sections of code are optimised.

2.2.2 Hardware dynamic optimisation

Many hardware based dynamic optimisation techniques work by optimising instructions in a *trace cache*. A trace cache is an instruction cache that stores traces of instructions that represent the order in which they are executed dynamically as opposed to how they are arranged in memory [RBS96]. Trace caches, when combined with branch prediction techniques, allow multiple basic blocks to be fetched simultaneously thereby improving fetch bandwidth for superscalar microprocessors. Traces are constructed by a hardware unit which, in its simplest form, is a buffer that accumulates a trace of instructions as they complete and then writes this trace into the cache. Traces do not need to match the representation of instructions in memory, allowing optimisations to be performed. For example, Intel's NetBurst architecture [HSU+01] does not store the original x86 CISC instructions in the trace cache; instead, instructions are partially decoded into a RISC-like representation before being cached. This allows the execution core to be a fast, simple RISC processor, allowing improved performance.

Trace cache fill units can be used as an alternative to a superscalar engine as a method of extracting parallelism from a stream of sequential instructions [FS94]. The DIF machine [NH97] translates traces of instructions to execute in an internal VLIW parallel processor. The first time a trace is encountered it is executed in a scalar microprocessor and simultaneously scheduled and translated into a VLIW representation. Scheduled traces are then cached for future execution, removing the need to determine parallelism each time they are executed as in a superscalar microprocessor.

RePLay [PL01] extends the trace cache model by constructing regions of code containing no control instructions, called *frames*. The increased predictability gained from reducing the number of control instructions increases the efficiency of the fetch unit and reduces the need for fast branch predictor units. Frames are constructed by building long single-entry single-exit instruction streams consisting of many basic blocks, linked by highly biased branches. Initially code is executed as normal, and retired instructions are sent to the frame constructor. The frame constructor looks at the recent history of branches; once a branch has targeted the same address consecutively a certain number of times it is promoted to an assertion instruction. The basic blocks linked by this assertion are chained together into an instruction trace. Once this trace reaches a given size it is optimised and built into a frame. The created assertion instructions replace the conditional branches in the original instruction stream. Assertions check the conditions of the original branch, and fire if the expected conditions are not true. If an assertion fires, architectural

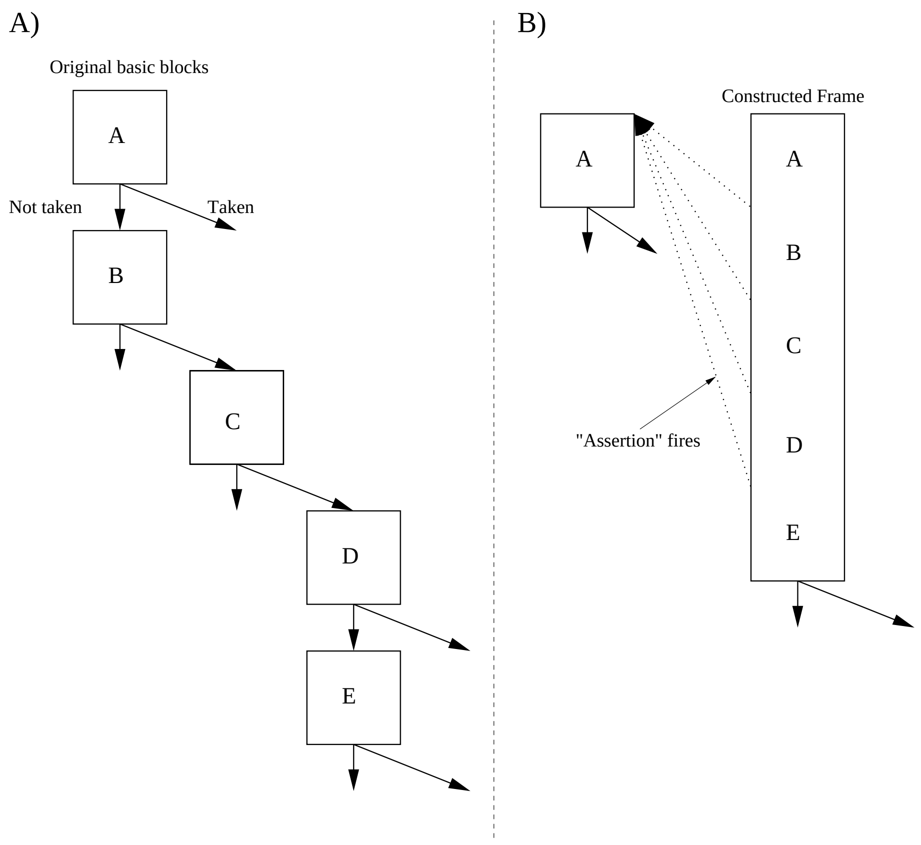


Figure 2.3: RePLay frame construction. A) Original basic blocks. B) RePLay frame, with conditional branches replaced by assertions. [PL01]

state is rolled back and control returns to the original, unoptimised basic block at the start of the frame (see figure 2.3).

Other systems perform more complex optimisations on the instructions in the trace cache. An Instruction Path Coprocessor (I-COP) [CS00] is an on-chip coprocessor that can be programmed to perform optimisations on instructions as they are added to the trace cache. The advantage of the I-COP is its programmability: a single simple piece of hardware can perform many different types of optimisation, and the optimisations performed can be customised for different applications.

Dynamic profiling can also be used to adapt the hardware for more efficient execution of the current program. By monitoring the relative frequency of operations in the trace cache, reconfigurable functional units within a superscalar processor can be adapted to provide more functional units able to execute the most frequent operations [NZ04].

Many of the above systems optimise or translate all code as it is encountered. Optimised sequences are stored, for example in a trace cache, for future re-use; any overheads due to performing optimisations will usually be overcome by repeated re-execution of optimised code. However, in many programs only a small portion of code is executed very frequently; typically 90% of instructions executed dynamically come from 5-10% of the code [HP]. Many sequences are only executed infrequently; therefore optimising these sequences may incur a greater overhead than the optimisations gain. The system proposed in this thesis reduces the optimisation overhead by performing *hot spot detection* in hardware to detect the most frequently executing sections of code and only optimise those, leaving the infrequently executed remainder of the code unoptimised.

2.2.3 Detecting hot spots

By monitoring a program as it executes a profile can be constructed; this can be used to determine which sections of code are worth optimising. Many profiling systems work by monitoring each branch in an executing program to give an accurate picture of how frequently sections of code have executed [CPMC96][CMH96][CLCG00]. The problem with this, from a dynamic optimisation viewpoint, is that it does not take into account how recently sections of code have executed; it merely gives an average of execution patterns in the program so far. This may cause changes in a program's behaviour to go undetected for a significant length of time. Code sections that execute very frequently for a short period of time may not be detected and optimised.

An effective hot spot detector must be able to react quickly to changes in program behaviour and predict which sections of code will be executed frequently. An accurate profile of the entire program's execution is not required as execution in the near future is likely to follow recent program behaviour. In addition, only frequently executed branches need to be profiled to detect hot spots. Duesterwald et.al. demonstrate that gathering accurate dynamic profile information is not necessary to effectively predict hot paths [DB00].

One method proposed by the IMPACT group is to use a modified branch target buffer (BTB) to monitor execution frequencies of branches [MTG$^+$99]. When a branch execute count exceeds a threshold a *candidate* bit is set. This table is cleared periodically of non-candidate branches leaving only branches which have executed frequently in that period. Candidate branches are then monitored; if a high enough proportion of the set of recently executed branches are candidate branches then the set of candidate branches is determined to be a hot spot. This hot spot is then laid out in the most frequently executed order in a region of memory called a *code cache*, and loop unrolling and other optimisations are performed [MTN$^+$00]. The traces generated are longer than in a standard trace cache and contain sequences of instructions that are more frequently executed in sequence as they have been ordered according to profile information, providing better performance.

The system described in this thesis proposes a hot spot detection mechanism, described in Chapter 3, based on monitoring the execution of basic blocks. A dynamic hot spot detection threshold, based on the number of currently active previously detected and optimised hot spots, allows the hot spot detector to vary its sensitivity based on the current behaviour of the program and react quickly to changes in that behaviour. The detector also monitors branch bias to allow the hot spots produced to consist of a chain of basic blocks linked with highly biased branches that has a single entry and exit point. This produces smaller hot spots than other systems such as the IMPACT system, but effectively removes control information from the hot spot, increasing the scope over which optimisations can be performed (see section 4.2). Detected hot spots are optimised and translated to execute in a parallel hardware execution engine.

2.3 Summary

Significant performance improvements can be gained by executing core algorithms of a program in reconfigurable hardware. These core algorithms are typically selected manually or by static profiling. As program behaviour may differ with different inputs these statically selected core algorithms may not match the most frequently executed code sections at run-time. Dynamic profiling techniques allow hot spots to be detected at run-time and therefore match the current program behaviour. This thesis explores the possibility of configuring hardware at run-time from object code by using dynamic profiling to select program hot spots.

Chapter 3

Hot spot detection

The architecture described in this thesis contains a reconfigurable hardware execution engine as part of a standard microprocessor. The most frequently executed sections of code, or *hot spots*, are executed in the reconfigurable hardware and the remainder of the code is executed by the microprocessor. This chapter describes the process of profiling the code to detect these hot spots.

3.1 Program behaviour

To justify attempting to exploit hot spots to improve performance it is first essential to analyse the run-time behaviour of example programs and determine the properties of any contained hot spots. Such properties include the number of hot spots, their size, how frequently they execute, what proportion of the program is executed within each and the lifetime of a hot spot within the duration of a program's execution. This can be performed by analysing a *dynamic trace* of a benchmark program. A dynamic trace is a listing of the order in which instructions are executed during the execution of a program. It differs from a static listing of a program in that instructions are listed each time they are executed and so instructions can appear multiple times; instructions that are never executed do not appear in the dynamic trace.

A dynamic trace is useful for analysing program behaviour as it contains information on how frequently sections of code are used and at what points during a program they are executed. This analysis can be simplified by looking at logical blocks of code instead of individual instructions; a suitable block of code is the *dynamic basic block*.

3.1.1 Dynamic basic blocks

A *dynamic basic block* is defined in this thesis as a maximal linear section of code with single entry and exit points and no internal control flow instructions such as branches or subroutine returns. They are dynamic entities: each instruction fetched immediately following a control flow instruction marks the entry point of a dynamic basic block; the exit point will be the next control flow instruction fetched. Dynamic basic blocks may overlap; overlapping dynamic basic blocks may have different entry points but will all end at the same control flow instruction. This can occur when a branch targets an instruction contained within an existing dynamic basic block: a second dynamic basic block exists at this entry point which overlaps the dynamic basic block containing the entry point (see figure 3.1). Note that using the standard static definition of a basic block here would result in two non-overlapping basic blocks divided at the point of the entering branch target. Using dynamic basic blocks simplifies run-time detection of entry and exit points because dynamically it is difficult to determine whether an instruction is targeted by a branch. For the remainder of this thesis *dynamic basic blocks* will be referred to simply as *basic blocks* for brevity.

Basic blocks are attractive from a code analysis and optimisation point of view for a number of reasons:

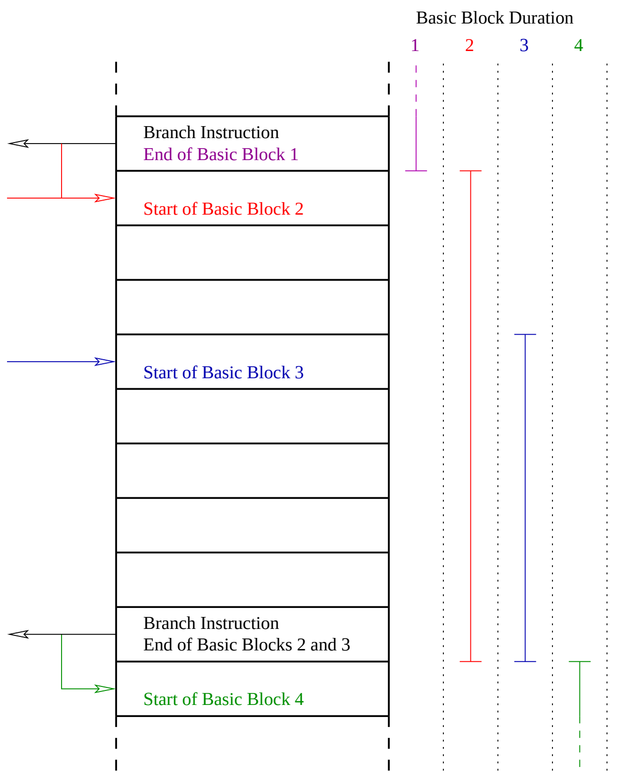


Figure 3.1: Layout of dynamic basic blocks in memory

Atomic nature As a basic block contains no control flow instructions each instruction within is executed precisely once each time the block is executed (ignoring conditional execution of instructions found in some instruction sets, for example ARM); the reverse is not true as an instruction can be contained in more than one overlapping basic block. This simplifies analysis of code execution as fewer points need to be monitored to obtain an accurate dynamic trace of a program.

Simple dynamic detection Basic block entries are straightforward to determine dynamically as a basic block entry point always follows the execution of a control flow instruction. A list of all targets of control flow instructions is equal to a list of all basic blocks.

Linear control flow The linear flow of instruction execution through the block enables simple determination of data dependencies when code is being

Benchmark	blowfish	cjpeg	nasm	qsort
Unique Basic Blocks	344	1155	2620	488
Total Basic Blocks Executed	1922283	12946104	3340064	4967372
Mean Executions per Unique Block	5588	11209	1275	10179

Table 3.1: Unique basic blocks and total number of basic blocks executed, dynamic trace

optimised. The operations performed by a basic block can be described by a linear data flow graph which can be used to optimise or parallelise the code within the basic block.

3.1.2 Dynamic analysis of executed basic blocks

A program contains many basic blocks; some of these will execute only once during the lifetime of the program whereas some will execute many times. A *unique basic block* is defined here as a basic block at a particular address that is executed at least once. The number of unique basic blocks and the total number of blocks executed during a run of some benchmarks is shown in table 3.1.

The size of the set of unique basic blocks varies considerably between benchmarks: during the execution of *blowfish* and *qsort* under 500 unique basic blocks execute whereas *nasm* has over 5 times this number. This corresponds to the algorithms involved: *blowfish* and *qsort* are performing a repeated small set of operations whereas *nasm*, an x86 assembler, performs a much wider variety of operations. The total number of basic blocks executed, and therefore the mean executions per block, also vary widely between the benchmarks, however these values depend on the size of the input. A larger volume of input will cause the number of basic blocks executed in these benchmarks to increase, with little or no increase in the number of unique basic blocks executed.

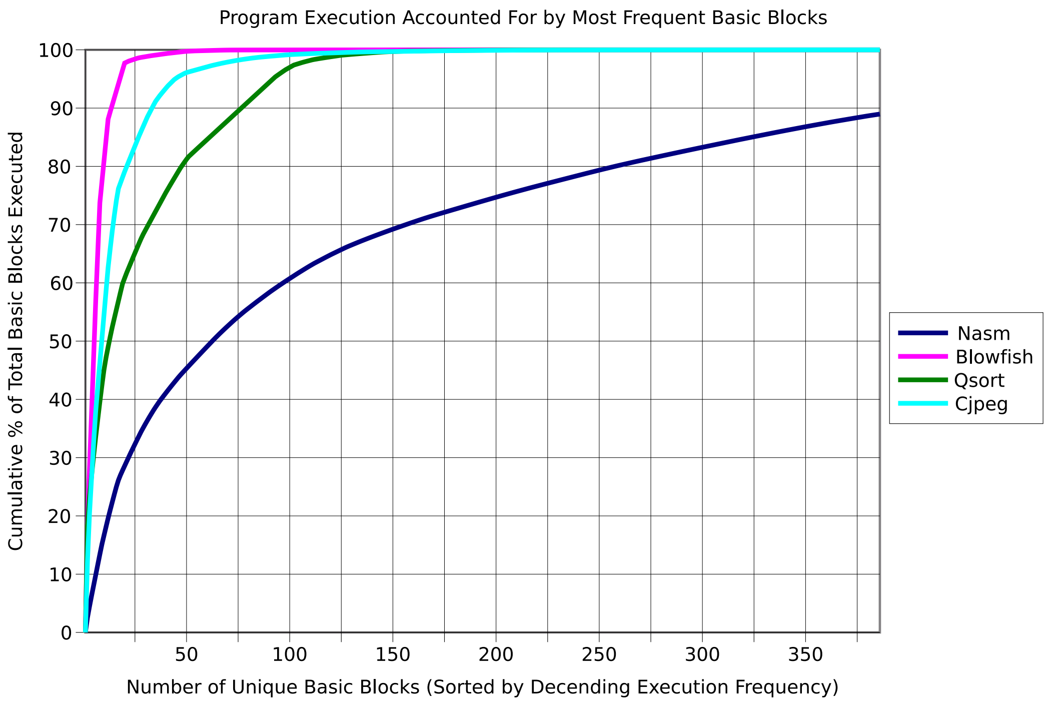


Figure 3.2: Cumulative percentage of total basic blocks executed by the most frequently executed unique basic blocks

3.1.3 Hot spots

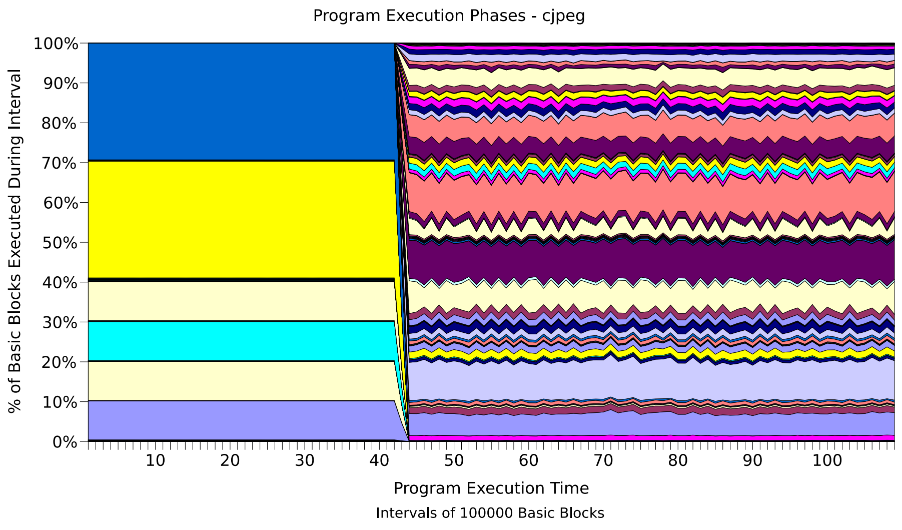
A hot spot is a section of code that executes frequently enough to take up a significant proportion of the total execution time. To determine whether hot spots exist, the number of times each unique basic block executes in a dynamic trace must be counted. The proportion of the blocks executed that is accounted for by the most frequently executed unique basic blocks can be used to show the presence of hot spots, as is shown in figure 3.2.

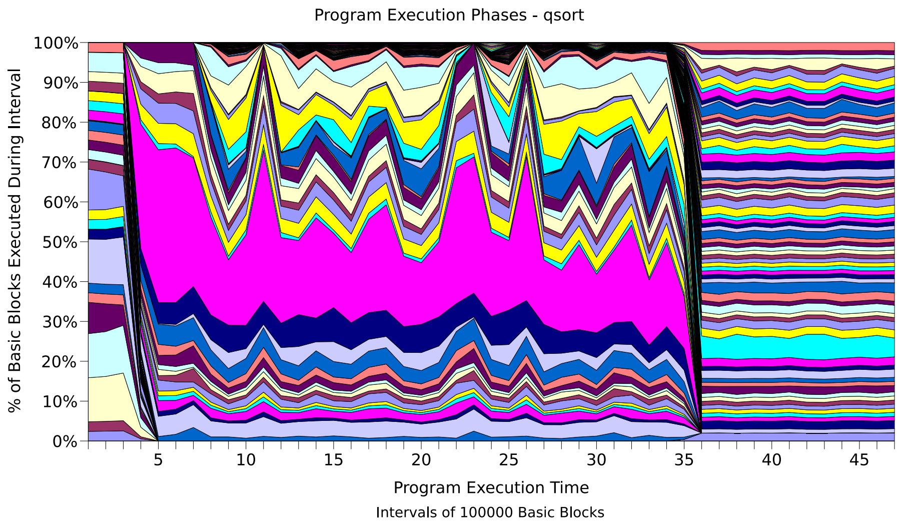
From this it can be seen that a small number of unique basic blocks accounts for a large proportion of the total dynamically executed blocks. The most frequent 16 blocks in *blowfish* account for 95% of the total blocks executed. *Blowfish* is an encryption algorithm and so performs a relatively small linear set of operations repeatedly on an input file, accounting for the small working set of basic blocks. Compared to *blowfish* the most frequent blocks in *cjpeg* and *qsort* account for a smaller percentage of the total blocks. This is partly due to the more complex algorithms involved (requiring more basic blocks within the main program loops) and partly because these programs exhibit *phased behaviour* (see next section). *Nasm*, an x86 assembler, performs very differently having a large working set of basic blocks, few of which execute very frequently. Over 60 of the most frequent blocks are required to account for 50% of the total blocks executed. This is because an assembler performs a large variety of operations on a relatively small set of input data. These results confirm a well known observation that often a large proportion of the executed time is spent within a small portion of the code [HP].

3.1.4 Phased behaviour

Many programs go through a number of stages, or *phases*, during execution. These phases correspond to different stages in the program. For example a program to encrypt a file first goes through an initialisation phase where memory is allocated, command line arguments are processed and files are opened, then enters a phase to generate the keys to be used to encrypt the file and finally, performs the encryption of the file itself. These phases typically involve differing program behaviour and each will require a different set of basic blocks containing different hot spots. The initialisation phase in the encryption program will likely be short and perform a variety of tasks, so will therefore contain few significant hot spots. The file encryption stage, however, will perform a very repetitive set of operations and therefore will contain a small set of very frequently executed basic blocks.

This behaviour can be seen in figure 3.3. The frequency for each unique basic block is shown, recorded for each 100000 basic blocks executed. Each horizontal slice represents a unique basic block. Different phases of execution are clearly visible in *cjpeg* and *qsort*. The working set of frequent blocks can be seen to change about two fifths of the way through the execution of *cjpeg*. Almost 60% of the total basic block executions in the first phases are from two





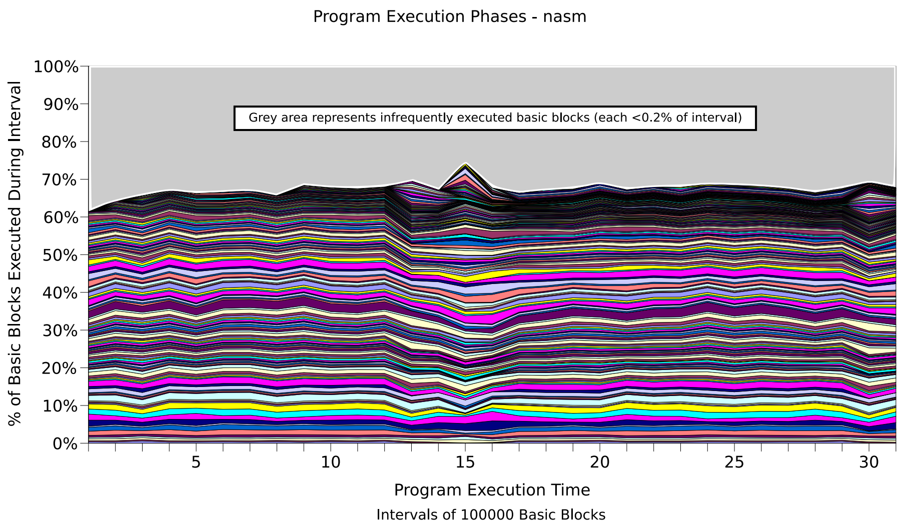


Figure 3.3: Phases during program execution: each horizontal slice shows the execution frequency of a unique basic block as a percentage of the total basic blocks executed during each time interval

basic blocks; most of the remaining 40% are from only four more. These blocks make up a significant hot spot in this first phase. *Qsort* has three phases: reading in the unsorted file, performing the sort itself and then outputting the sorted data. Unlike these two benchmarks, *nasm* does not show distinct phase changes, although there are slight changes in behaviour about half way through and towards the end of the program. No horizontal slice on this graph is significantly thicker than the rest, indicating that there are no significant hot spots in this benchmark. The working set of basic blocks in this benchmark is much larger than in the other benchmarks; the grey area at the top of the graph represents unique basic blocks that are too infrequent to show (unique blocks that make up less than 0.2% of the total blocks during an interval).

The presence of phased behaviour suggests that for this system to take full advantage of the hardware execution engine it should be able to adapt to the current program behaviour. This requires the currently running program to be monitored at run-time to determine the current hot spots and to detect when these hot spots change. Section 3.4 proposes a mechanism to achieve this.

3.2 Chains of basic blocks

So far it has been assumed that hot spots would consist of single basic blocks. However basic blocks are typically small, averaging about seven instructions in length (see figure 3.4). Compiling these into hardware does not give much scope for optimisation and parallelism within the blocks and the short length of blocks will increase the frequency and therefore the cost of switching between hardware and software.

Blocks to be compiled into hardware can be increased in size by chaining together several that are usually executed in sequence. If the branch (or other control flow instruction) at the end of a basic block is taken in the same direction a high proportion of the time then it is determined to be a *highly*

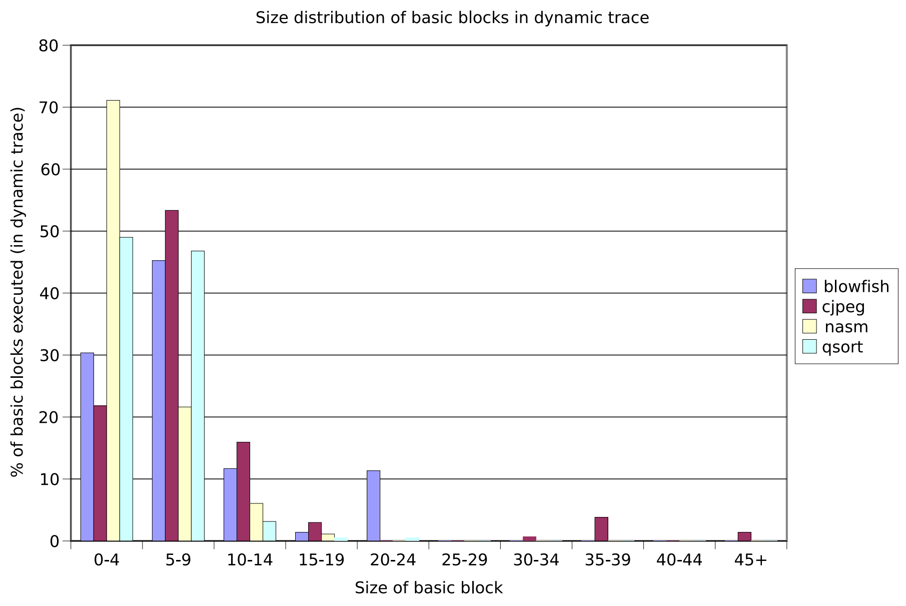


Figure 3.4: Distribution of basic blocks in dynamic trace by number of instructions

biased branch. Basic blocks linked by highly biased branches can be linked together to form a larger block of instructions or *block chain*. Block chains can be constructed by repeating this process until a block is found with a branch that is not highly biased, a branch is found that loops back into the chain or the chain becomes too large. Similar methods are used in the DIF [NH97] and RePLay [PL01] mechanisms to improve optimisation scope.

The chain of blocks can be treated as a single basic block as it is compiled: all contained branches are removed and the basic blocks become a single linear block of instructions containing no control flow. During execution the situation may arise where a highly biased branch is taken against its bias; this is treated as a special case and is discussed in section 3.2.1.

Increasing the size of the compiled blocks brings a number of benefits:

Improved optimisation scope. A greater number of instructions within the chain increases the scope for optimisations to be applied. Removing branches between the blocks removes control dependencies, increasing the number of independent operations and therefore increasing available parallelism. Some operations may be redundant due to not-taken branches and can be removed, and some operations for control flow (for example branch calculations and procedure call stack operations) can also be removed from the block. Section 4.2 discusses these advantages in more detail.

Reduced switching between hardware and software. Larger blocks will run for a longer period of time and, as there are fewer of them to cover a given amount of code, they are less frequently entered reducing any overhead from switching between hardware and software execution. In addition the number of temporary values within the block will be increased thus reducing register bank accesses (see section 4.1.1).

Reduced number of hardware compilations. Fewer individual compilations will reduce the number of compiled block definitions that must be cached and reduces the overhead of maintaining this cache (for example deciding which compilation to reject upon a new entry being added). The number of compilations performed will be reduced, although the compilation time will increase with larger, more complex blocks.

3.2.1 Handling unexpected branch decisions

When block chains are executed in hardware they are treated as a linear block of instructions containing no internal branches. Chaining together blocks linked by unconditional and deterministic branches is straightforward: the branch will always be taken and so it can simply be removed. If, however, the branch is conditional the condition must be tested. The majority of the time the result of the condition test will match that observed during profiling and the block chain will complete as expected although the case may arise when the test would cause a branch out of the chain of blocks. This case must be handled to ensure the program executes correctly. It is expected that these *breakouts* are infrequent due to the highly biased nature of the branches and so handling them can be slow without significantly affecting the overall performance of the system. Chain breakouts could be handled in the following ways:

Continue executing in software from breakout point. Upon detecting that a breakout has occurred the hardware structure transfers results generated before the breakout point to the microprocessor and execution continues in software. This requires the hardware structure to be able to hand control back to the software after each basic block, requiring values in each architectural register to be synchronised at the end of each basic block in the chain. This reduces the ability for operations to be scheduled independently of their original basic block, reducing the scope for parallelisation. Alternatively, register values can be preserved for each possible breakout point using register renaming [NH97], increasing the number of registers required, and operations in basic blocks before a possible exit point must be executed before the breakout can occur.

Restore state to before chain began executing. If a chain must be broken out of, all results generated in the reconfigurable hardware are discarded and architectural state is restored to the point before the chain began executing. Control is handed to the microprocessor and the chain executes from the start in software. This requires the state before the chain began executing to be preserved until it is known that the block will complete. Values can only be committed to memory and registers after it is known no breakout can occur.

The latter is expensive when a breakout occurs as results generated so far have to be discarded, but this is very infrequent (see section 6.2.2) and so overall this cost is expected to be small. Therefore, this method is used, as control dependencies between operations in different basic blocks can be removed, allowing them to be scheduled in parallel if no data dependencies exist (see section 4.2.2). Additional hardware is required to preserve architectural state while the chain executes. Values in registers can be preserved by, for example, using register renaming [Kel75]: two copies of each architectural register exist, one which preserves the value at the start of the chain and one which is written to by the hardware structure. These are switched to the original values if the chain is broken out of; if it completes fully the values written by the block executing in hardware are used. A similar mechanism exists in Transmeta's Crusoe microprocessor [DGB+03]: speculatively executed operations write to a working copy of a register whilst the original value in the register is shadowed in an additional register. These shadow values can be copied into the working registers if one of the assumptions speculated upon proves to be incorrect, restoring the original state.

A write buffer [MTL95] can be used to hold stores to memory while the block is executing; this can be flushed back to memory once it is known that the block will complete. Section 5.3.1 describes this hardware support in more detail.

3.2.2 Change of branch behaviour

The situation may arise when the behaviour of a branch changes during the course of a program's execution. A branch that was previously highly biased may lose this bias or it may be reversed. If this branch has been compiled into a block chain this will cause the chain to break out frequently, adversely affecting performance. This is resolved by counting the number of times a chain is broken out of; if this count exceeds a threshold then the compiled chain can be discarded, possibly allowing the hot spot detector to re-detect and recompile it with the new branch biases. Section 3.6 details how branch bias change is detected.

3.2.3 Increase in block size

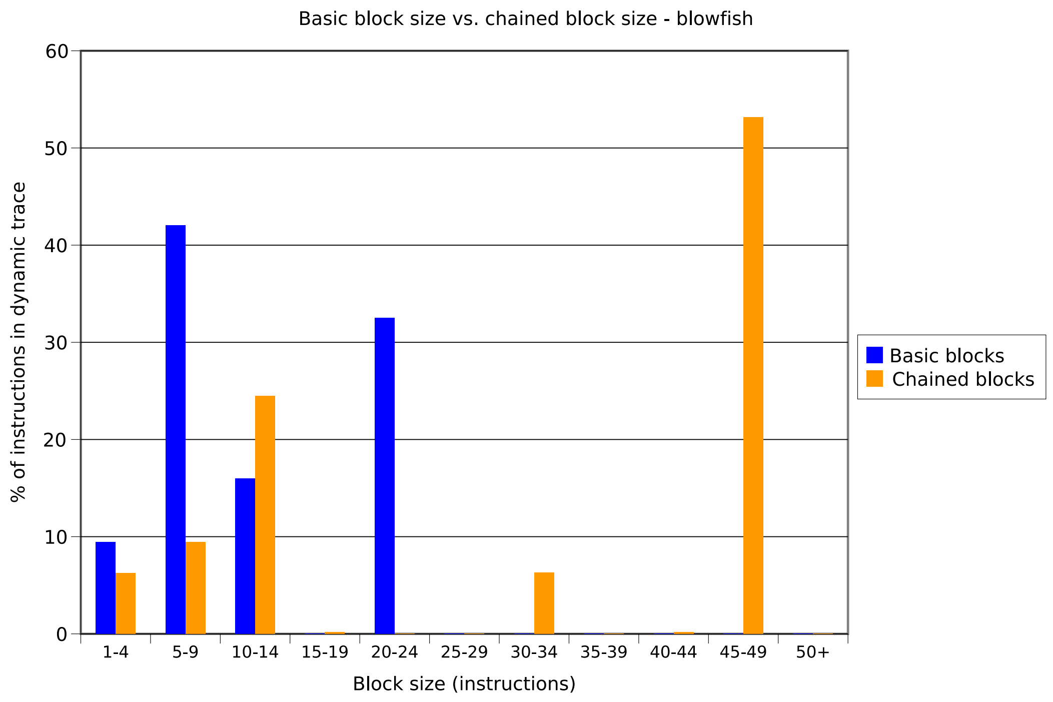
Figure 3.5 shows the proportion of instructions (in a dynamic trace) contained within different block sizes and compares basic blocks and chained blocks for *blowfish* and *nasm*. Blocks were chained together if the branch bias was $\geq$99% in one direction. In *blowfish* the block chaining process more than doubles the size of the blocks in hot spots. The increase in *nasm* is less significant, although chaining does reduce the number of instructions that execute in blocks fewer than 5 instructions in size by about a third. This suggests that there are few highly-biased branches in *nasm* compared to *blowfish*; this supports other observations made that indicate the behaviour of *nasm* is not as predictable as the other benchmarks.

3.3 Profiling

For frequently executed basic blocks to be compiled into hardware they must be detected to be part of a hot spot. To detect hot spots the code must be *profiled*. Profiling is either *static*, when it is performed in advance, or *dynamic* when it occurs at run-time.

3.3.1 Static profiling

Static profiling is performed in advance of the program being executed, usually during the compilation process of the program. The program is executed in a simulator with a sample set of data and a trace is produced. This trace is used to determine the most frequently executed code sections within the program which are then optimised and compiled into reconfigurable hardware configurations. These configurations are then distributed with the software binary, where they are loaded into the reconfigurable hardware at the appropriate time.



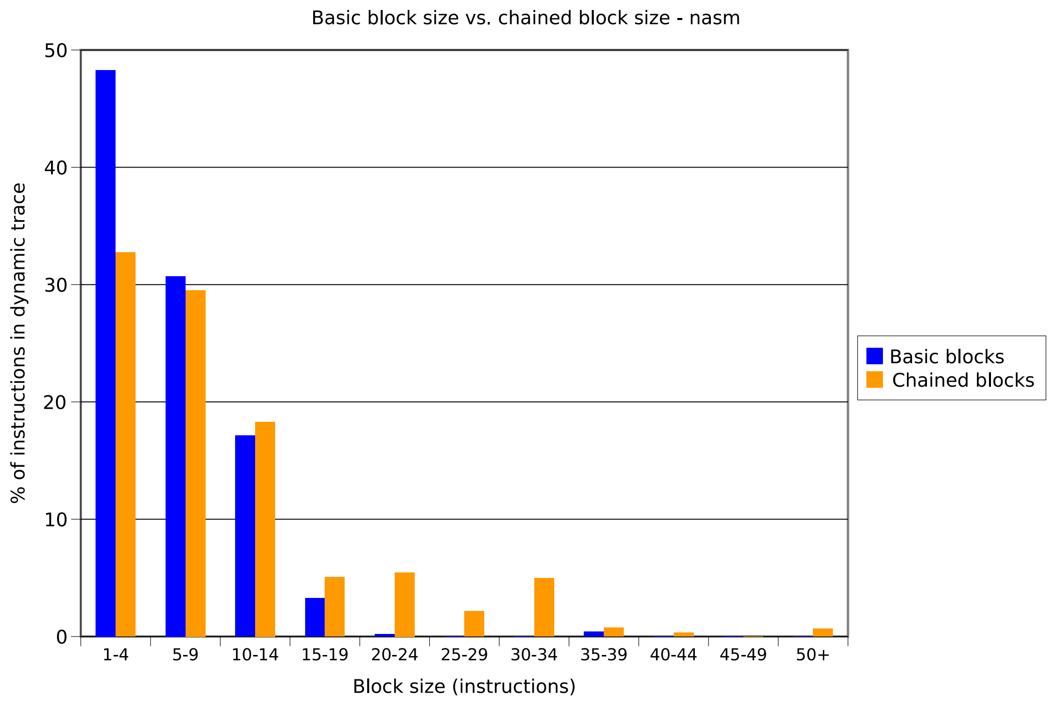


Figure 3.5: Basic block size vs. chained block size

The primary advantage of static profiling is that all the processing involved with profiling, optimisation and compilation is performed at compile-time and so there is no overhead at run-time.

However static profiling has significant disadvantages: the sample set of data used to generate the profile can differ from the data used at run-time. This can cause the program to behave differently from the static profile. Different options or program environments can also change its behaviour. This can change the hot spots in the program causing the hardware configurations generated by the static profile to no longer be valid.

With a static profile, configurations can either be loaded at the start of the program or swapped in and out of the hardware by specially inserted instructions. The latter requires phase changes to correspond to a particular point in the code. This may not be the case if behaviour changes are caused by changes in the data being processed, and so configurations must be available throughout the program even if they are not currently in use. This increases the storage required to hold configurations, possibly limiting the number of hot spots that can be compiled.

Another disadvantage of static profiling is that compilations must be distributed along with the program binary; this is not a problem for new code but legacy binaries must be profiled if they are to gain any benefit from the reconfigurable hardware.

3.3.2 Dynamic profiling

Dynamic profiling is performed at run-time: the code is monitored as it executes and sections that execute frequently enough to pass a threshold are compiled into hardware. Dynamic profiling looks at the currently executing instance of the program, therefore detected hot spots are correct for the current input data and program options. Changes in program behaviour during run-time (such as entering a new phase of execution) will be detected by the dynamic profiler and a new set of hardware configurations will be generated. The set of compiled hot spots can adapt to the current program behaviour, so the need to have configurations for hot spots in all phases to be available at all times is removed. This reduces the space required to store configurations, reducing the size of the hardware or allowing a greater number of currently active hot spots to execute in hardware.

To perform dynamic profiling a method of monitoring a program at run-time is required. One of the following methods can be used:

Use a software interpreter. The program is interpreted and profiled until hot spots are detected. Once detected, hot spots are executed in hardware and the remainder of the program continues to be interpreted. This requires no modifications to the code or the hardware; however, interpreting code is much slower than executing it on a microprocessor so any improvements from optimisation will have to overcome this overhead for a resulting net gain.

Inserting monitoring instructions. The program can be run on a standard microprocessor with no additional hardware. Code modifications to insert monitoring instructions to update a counter table in memory are performed before the program executes. These additional instructions, plus the need to periodically break out of the program to look for hot spots, will add delay to the executing program. Instructions must be added to each basic block to construct an accurate profile. If only an estimated profile is required then the number of added instructions can be reduced by only instrumenting blocks in strategic places (for example loop or procedure call entry points); however, the ability to profile individual branch bias would be lost.

This method requires altering the program binary before execution begins by inserting instructions and altering internal addresses such as branch offsets or instructions that load constants from the code space.

This method is difficult with dynamically linked libraries: they would either need to be altered in a similar fashion (requiring a copy to be created if a loaded library is shared between multiple programs) or ignored and not profiled.

Performing monitoring in hardware. No change to software is needed. Additional hardware is required to maintain a table of monitored points. This allows monitoring to be performed in parallel with the executing program thereby not adversely affecting performance.

As the system proposed in this thesis already requires new hardware to support the reconfigurable execution engine, the main disadvantage of the third method is removed. Therefore this option is used in this system as it incurs no performance penalty and requires no software modification.

3.4 Operation of hot spot detector

The hot spot detector hardware must maintain a table of how frequently each section of code, in this case each basic block, is executed over a period of time. This table could be be stored in main memory, which would allow it to be very large but would require a memory cycle each time it is updated or monitored, reducing performance. Storing it in a small on-chip memory would not affect performance but would limit the number of basic blocks that can be monitored. However, as only frequently accessed blocks need to be monitored, the table is maintained as a cache with a replacement policy that discards the least "hot" blocks, reducing the amount of storage required to produce a useful profile of the most frequently executed blocks.

In the system described in this thesis a table of recently executed basic blocks is maintained on chip in a *block profile table* (BPT). Each basic block entry in the BPT contains a counter which records the number of times the basic block has executed in the recent past, or how "hot" the block is. If a counter exceeds a threshold the compiler is invoked to create a hardware configuration for that block. The structure and operation of the BPT is detailed in section 3.5.

Once the detected hot spots have been compiled (see section 4.3), the hardware configuration information is stored in a *hardware configuration table* (HCT) where it can be loaded on demand into the reconfigurable hardware structure. The basic blocks contained in the compiled chain are removed from the BPT. Only basic blocks that execute in software are monitored by the BPT to prevent already compiled blocks from being detected and compiled again. The HCT is described in more detail in section 3.6

3.4.1 Determining which blocks to compile

For a block to be worth compiling into hardware it must be expected to execute frequently in the future. This can be estimated by looking at the block's history: if a block has executed frequently in the recent past it is likely to continue doing so. Blocks that execute very frequently in a short period of time are probably part of an inner loop that iterates many times: these blocks should be compiled as soon as possible to allow the remainder of the loop iterations to execute in hardware. If a block has executed regularly for a long period of time then it is likely to continue doing so and should also be compiled. However if a block has executed frequently but not for a long period of time then it is unlikely to execute again and so should not be compiled. Blocks that only execute occasionally should also not be detected as hot spots.

For the hot spot detection mechanism to perform in this way it must apply a higher weighting to recent executions of a block than older executions. This can be achieved by periodically decrementing all the counters in the table to *age* them. This has the effect of increasing the effect of recent counter increments relative to older ones, thereby increasing the importance of recent block executions in determining hot spots. Counters for blocks that begin executing very frequently will increase rapidly, allowing quick detection and compilation. Any block entry that does not execute for a long period of time will have its counter gradually decreased until the cache replacement policy rejects it from the profile table. Counters for blocks that do not execute frequently enough will not break the threshold and be detected as hot spots. The mechanism to achieve this is described in setion 3.5.

3.4.2 Sensitivity of the hot spot detector

The setting of the increment size, hot spot detection threshold and rate of ageing affects the *sensitivity* of the hot spot detector. This sensitivity determines how frequently a block must execute during a given time period to be detected as a hot spot and be compiled. Ideally, the sensitivity should be set to keep the HCT full with configurations for the current most frequently executed hot spots. A high sensitivity (with a high increment rate, low threshold and low rate of ageing) will decrease the delay between a block becoming hot and the block being detected as a hot spot, ensuring the HCT is filled quickly at the start of a program or upon a program phase change. However, this will also increase the number of block compilations, each of which must replace an existing compiled block in the HCT. This is undesirable as a cost is associated with performing a compilation and so compiled blocks should be kept for as long as they remain in use, unless new very frequently executed blocks are detected. A sensitivity set too high may also lead to thrashing in the HCT if rejected blocks are quickly re-detected and recompiled. Conversely a sensitivity set too low will cause few blocks to be detected, causing the reconfigurable hardware to be under-utilised.

The setting of the hot spot detector sensitivity is further complicated by variations in the number and frequency of hot spots in different programs. Some programs contain more simultaneously active hot spots than available locations in the HCT, leading to thrashing if the sensitivity is too high. Other programs may contain a large number of blocks, none of which is executed frequently enough to exceed the threshold unless a high sensitivity is used, causing the reconfigurable hardware to remain idle. Furthermore behaviour may change during a program as different phases are entered. The choice of sensitivity setting is therefore dependent on the current program behaviour.

This variability means a fixed, pre-set sensitivity is unlikely to provide optimal performance in many cases. Figure 3.6 demonstrates this. These results were generated using a software model of the hot spot detector, with both the BPT and HCT having 64 entries (BPT and HCT size are discussed later in this chapter). The first graph shows the percentage of blocks detected as hot spots and executed in hardware with different sensitivities. The sensitivity is varied by altering the amount the recent execution counter in the BPT is incremented when a basic block is executed; the rate of ageing and the threshold remain constant. The proportion of blocks executed in hardware is highest with a sensitivity of around 256-512 in all benchmarks except for *nasm* which peaks at 128 and then drops as the sensitivity is increased further. The second graph shows the number of compilations during each benchmark's execution. This increases at a different point for each benchmark as the sensitivity increases. *Nasm* and *cjpeg* show sharp increases with no corresponding increase in the percentage of blocks executed in hardware, indicating that thrashing in the HCT is occurring. The third figure shows an estimated cost function as a percentage of executing entirely in software (this is based on a block executing in hardware costing 40% of executing in software and a compilation cost of executing 1000 software basic blocks). The minimum cost is at a different sensitivity for each benchmark, showing that having a fixed, pre-determined sensitivity would not give optimal performance in all cases.

A solution to this is to vary the sensitivity depending on the current number of active compiled hot spots. A *recently used* flag is added to each loaded block configuration in the HCT. This is set when a block is executed (or has just

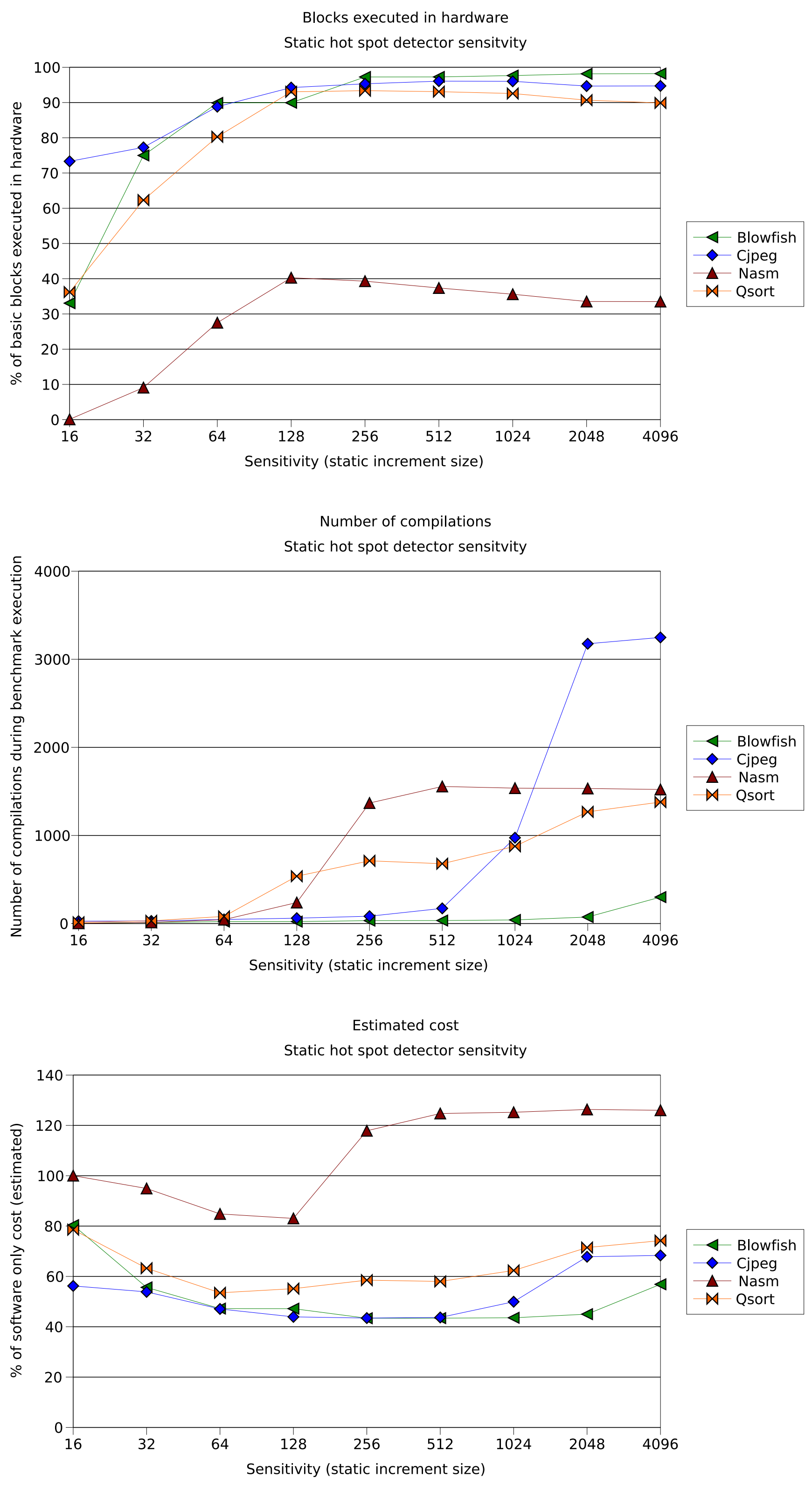


Figure 3.6: Number of blocks executed in hardware, number of compilations and estimated cost with static hot spot detector sensitivity

been compiled) and is cleared if a block does not execute for a predetermined period of time (see section 3.6). This information can be used to increase the sensitivity of the hot spot detector as the number of inactive locations in the HCT increases. When all the blocks in the HCT have been used recently the hot spot detector sensitivity will be very low, requiring any new basic blocks to execute very frequently to be detected as hot spots, preventing blocks currently in use from being replaced. If the HCT is empty, as it is at the start of a program, then the sensitivity is set very high to fill the table as quickly as possible and begin utilising the reconfigurable hardware. Upon a program phase change the previously compiled blocks will become inactive, increasing the sensitivity of the hot spot detector and allowing hot blocks in the new program phase to be detected and compiled quickly.

Figure 3.7 shows a similar graph to figure 3.6 with a sensitivity that changes with the number of slots in the HCT that do not contain an active block configuration. The numbers on the x-axis are the base sensitivity: this is a multiplier of the number of inactive locations in the HCT to give the current hot spot detector sensitivity. The number of blocks executed in hardware remains approximately constant over a wider range of values than the static sensitivity. The number of compilations remains low even as the base sensitivity increases; this is in contrast to a static sensitivity where the number of compilations increases rapidly as the sensitivity increases. This shows that the dynamic sensitivity is effective at preventing thrashing. The third graph combines the proportion executed in hardware and the number of compilations using the same cost function as in the static sensitivity cost figure. This shows a shallow bathtub curve, with the minimum point in each benchmark aligning at a base sensitivity of between four and eight. This is a significant improvement over the cost graph with static sensitivity where minimum points are narrower and do not align between the benchmarks.

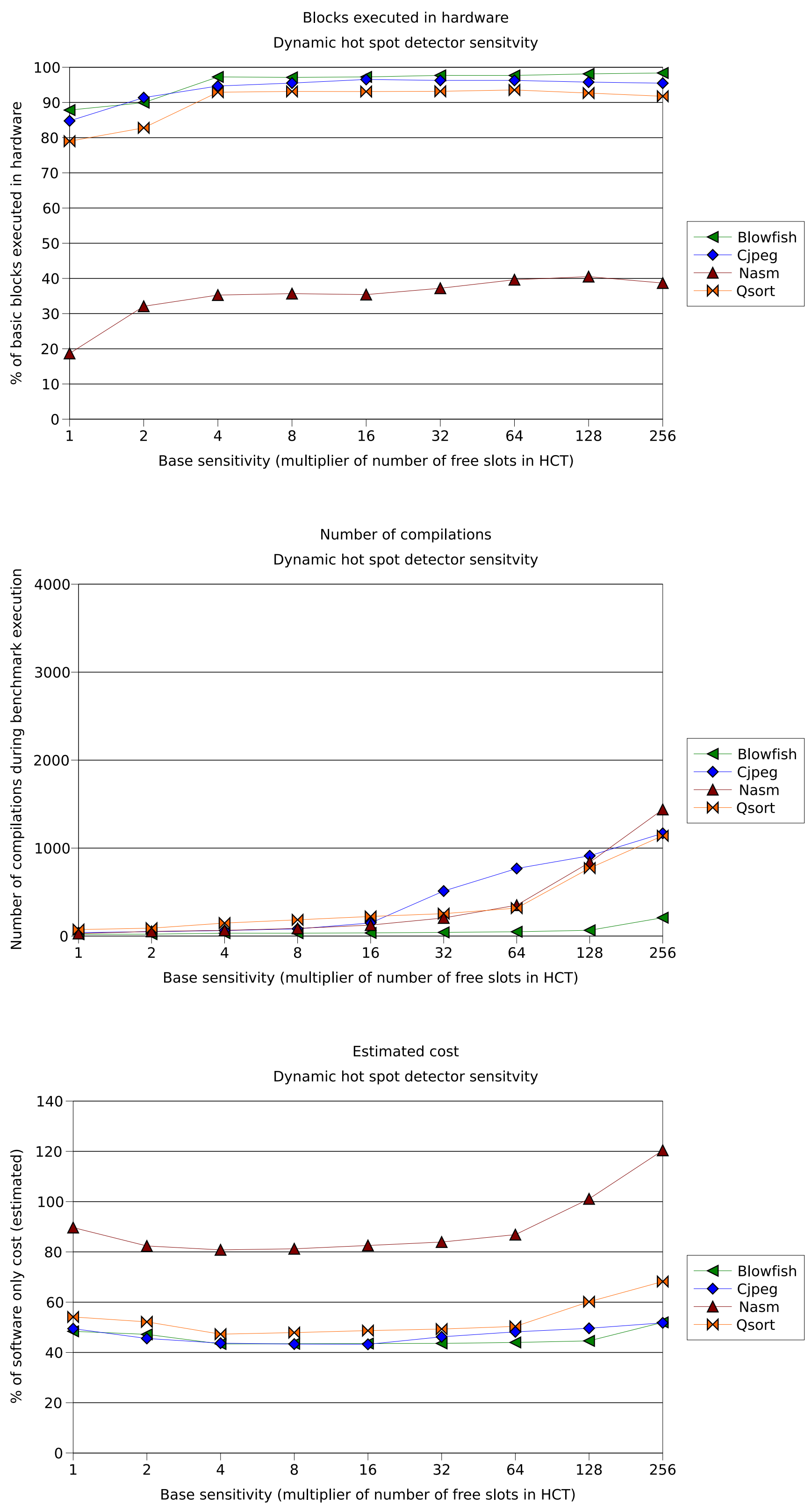


Figure 3.7: Number of blocks executed in hardware, number of compilations and estimated cost with dynamic hot spot detector sensitivity

These results show that a dynamic sensitivity based on the number of active compiled blocks can provide an effective method of automatically regulating the hot spot detector, and the wider, flatter curves in the cost graph show dynamic sensitivity requires less fine-tuning of settings than static sensitivity. This mechanism allows the hot spot detector to adapt to radically different code and to quickly detect changes in program behaviour.

3.4.3 Constructing Block Chains

To perform block chaining the bias of the branch at the end of a basic block must be monitored. Each entry in the BPT contains two counters: one records the number of times the block has executed, and the other counts the number of times the branch at the end of the block is taken. These are used by the compiler to determine whether the branch is highly biased. If so, the block branched to is added to the end of the original basic block to construct a chain of basic blocks. The branched-to block will also have been frequently executed in the recent past, and so is also likely to contain an entry in the BPT which contains branch bias information for the end of the block, which can be used to extend the chain further. This continues until either a non-biased branch is found, a block is branched to that is not in the BPT or the chain exceeds a maximum size, at which point chain construction is terminated and the chain is compiled into a hardware block and stored in the HCT. Section 4.3 describes the compilation mechanism in more detail.

Indirect branches such as returns cannot be accurately profiled with a binary taken or not-taken decision as the relative frequency of targeted addresses is not monitored. However, an approximation is made with returns: they are only permitted to be included in a chain if the matching call is also part of the chain, allowing the return address to be determined. This approximation fails if the return's taken or not-taken bias is different when the subroutine is called from different locations. However, this situation is likely to be infrequent and permits a significant simplification of the profiling mechanism. Chaining across returns is performed by matching link pointer values on the stack within the chain as the block is compiled: if the link value being restored to the program counter was added earlier in the chain then the call is entirely contained within the chain and so can be inlined. Chains are not constructed around indirect branches that are not determined to be returns for calls contained in the chain.

3.5 Block Profile Table

The Block Profile Table (figure 3.8) is responsible for monitoring basic blocks that execute in software. It constructs a simple profile of recently active basic blocks and an estimate of the branch bias at the end of each of those blocks. The table is structured as a fully associative cache indexed by the block address and with a replacement policy based on rejecting the least hot block when a new block is encountered. The table contains the following fields:

Block address This is the entry point of the basic block being monitored. Due to the way basic blocks are defined in this system (see section 3.1.1) this can be used to identify when the basic block has been entered by monitoring the next instruction fetched after each control flow instruction.

Aged block counter This provides a measure of how *hot* the block has been in the recent past. This value is increased each time the block is executed and is gradually decreased: this is done by periodically decrementing all block entries in the table. The size of the increase varies depending on the number of active compiled blocks in the HCT, as described in section 3.4.2. If the value in this field exceeds the compilation threshold the block is compiled and removed from the BPT. This field is also used to determine which block in the BPT to replace when a basic block not in

Block Address	Aged Block Counter	Total Execute Count	End Branch Taken Count

Figure 3.8: Block profile table

the table is executed. Results were generated by decrementing all entries by 1024 every 1024 software basic blocks: this reduces the number of subtract operations that must be performed. The value used for the block detection threshold when generating results was 8192.

Total execute count This is a simple counter that is incremented each time the block executes; it is not aged. This is used along with the *end branch taken count* field to determine the branch bias when the block is compiled.

End branch taken count This counter is incremented when the block is executed and the branch at the end is taken. When a block is entered a reference to the block's location in the BPT is maintained until the block exits, at which point the end branch taken count is updated according to whether the control flow instruction at the end was taken. This is used along with the *total execute count* field to determine the branch bias when the block is compiled. A branch with a bias $\geq$99% in one direction is determined to be highly biased and the basic blocks linked by the branch are chained together.

Blocks are removed from the table when a chain containing them is compiled. If the table is full and a new block is detected then the least 'hot' block is rejected.

3.5.1 Size of block profile table

The size of the block profile table determines how many basic blocks can be profiled simultaneously. This has an effect on which blocks are detected as hot spots. Figure 3.9 shows the percentage of instructions executed with different BPT sizes (with the HCT size fixed at 64 entries). Very few instructions execute in hardware with a single entry: this is expected, as the entry in the BPT would be replaced on each new basic block executing, meaning that only blocks that looped back to themselves would be able to break the hot spot detector threshold. Performance improves as the number of monitored basic blocks increases: maximum performance is achieved in *blowfish* and *cjpeg* with 16 entries in the BPT. This is due not only to the relatively small working set of basic blocks in these benchmarks, but also because the set of blocks in use does not change frequently. Performance in *qsort* continues increasing until the table size reaches 64 entries. This benchmark contains a larger working set of basic blocks that fluctuates frequently (see figure 3.3) requiring a larger number of basic blocks to be monitored to detect hot spots accurately. Execution time in *Nasm* is not dominated by a small number of hot spots like in the other benchmarks; its performance here is constrained by the HCT size of 64. This limits the percentage of executed instructions that can be covered by loaded configurations to about 30% (see figure 3.11); this value is reached at a BPT size of 64 and no further increases can be gained by increasing the size of the BPT further.

A relatively small table size of 16 is sufficient for programs that contain small, constant hot spots such as those that process data in a streaming, linear fashion. Programs that have less predictable execution patterns perform better with a larger table size of 64 entries. Increasing the BPT above this size has little effect with a limited HCT size. Therefore a BPT size of 32 or 64 is recommended for these benchmarks.

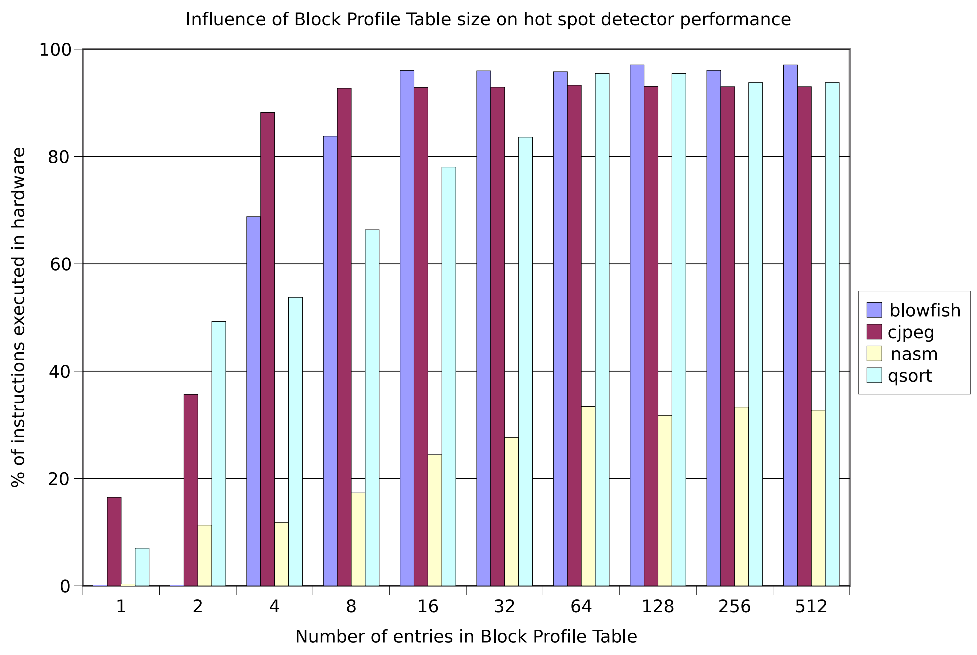


Figure 3.9: Influence of Block Profile Table size on number of instructions executed in hardware

3.6 Hardware Configuration Table

The Hardware Configuration Table (figure 3.10) stores data about the blocks currently compiled into hardware configuration, and is used to determine how many blocks are currently active. Like the BPT it is structured as a fully associative cache, indexed by block entry point address. It contains the following fields:

Block entry point address This is the address of the entry point into the hardware block. If the address of the instruction fetched after a control flow instruction (including after the exit of another hardware block) matches one of these the corresponding block is loaded into the reconfigurable hardware structure and executed.

Executed recently field This field records whether the block has been executed recently. When the block is executed the top bit in this field is set.

Bits are periodically shifted right which will cause the field to be cleared if the block is not executed for a period of time. The number of entries in the HCT for which this field is zero affects the amount the *aged block counter* in the BPT for a software block is incremented when executed. This field also provides a simple estimate of how regularly the hardware block has executed in the recent past by counting the number of bits set, and so is used to select a block configuration to reject when compiling a new block into a full HCT.

Breakout and breakout reset period counters The breakout counter monitors the number of times the block has broken out of early, due to a chained branch being taken against its measured bias or another exception during block execution. This is an expensive operation as it requires execution progress in the block so far to be discarded and re-executed in software (see section 3.2.1) and so needs to be detected and handled if it occurs too frequently. This counter is incremented when the block is broken out of. The value is cleared periodically; if it exceeds a threshold limit before this the block is deemed to have changed behaviour since it was compiled and so is rejected from the table. If the block is still sufficiently active it will be detected again by the BPT. The threshold used for the generation of results in this thesis is 64 breakouts per 1024 block executions. Increasing the threshold (or decreasing the reset period) will allow more breakouts to occur before a block is rejected. This has the effect of increasing the tolerance for blocks that occasionally break out, and so the number of breakouts will increase as fewer of these blocks will be rejected; conversely this is likely to increase the number of compilations as these blocks are likely to be detected as hot spots again after they are rejected. The setting of these threshold values is therefore a trade-off between the expected cost of a block breakout and the expected

Address	Executed Recently	Breakout Counter	Reset Period Counter	Hardware Configuration

Figure 3.10: Hardware Configuration Table

cost of a recompilation, including the time taken for the BPT to re-detect the hot spot and determine the new branch biases.

Hardware data This contains the location of the hardware configuration data in the configuration storage, for loading into the hardware execution engine when the block is entered.

3.6.1 Storage of hardware configuration data

The hardware structure described in section 5.3 is configured using microcode-like configuration data stored in an on-chip memory. All the configuration data for one execution cycle of the hardware structure must be loaded in parallel from this memory and used to configure the hardware structure for the cycle of execution. The width of this memory should therefore match the size of the configuration data for each cycle of execution (see section 5.2.5).

This configuration memory could be structured in a number of ways. The simplest would be to have a fixed-size block of memory for each entry in the HCT, each large enough to store configuration data for the largest possible compiled block. However, this results in an inefficient use of available memory when smaller blocks are compiled. A more complex method would be to have an additional memory structure separate from the HCT and allocate a block of memory for each cycle of configuration data in a compiled block. Each cycle's data would also contain a pointer to the configuration data for the next cycle in the currently executing block, allowing a chain of configuration data to be loaded as the block executes. This would require additional management of the available memory resources to monitor which locations contain valid configuration data for blocks loaded in the HCT, and to invalidate these when a block is rejected from the HCT. Such a memory management unit has not been developed as part of this thesis, and so the first method is used.

3.6.2 Number of entries in hardware configuration table

The HCT stores configuration information for compiled hot spots. This is used to configure the hardware execution engine when a compiled hot spot is executed. The size of the table determines how many of these compiled blocks can be stored simultaneously; once the table is full any newly detected and compiled hot spots must displace an existing configuration in the table. This is undesirable if it causes a hot spot to be compiled again, as a significant cost is associated with performing the compilation process. This table should therefore be large enough to hold configurations for all hot spots that are active in the same part of a program.

Figure 3.11 shows the effect of the HCT size on the percentage of instructions executed in hardware. Most of the benchmarks reach a maximum performance with a HCT size of 32 or 64 before levelling off; at this point all significant hot spots can be contained within the HCT so further increases in the size of this table have little effect. Again *nasm* performs differently, requiring a much larger table size to reach a high proportion of instructions executed in hardware as execution is spread dynamically over a large number of basic blocks and is not limited to a small number of hot spots. *Blowfish* performs well even with a very small HCT, demonstrating that a large proportion of dynamically executed instructions are contained within one or two very frequently executed regions of code.

These results have been generated using a hot spot detection mechanism with a dynamic sensitivity as was described in section 3.4.2. This mechanism will attempt to keep the HCT filled with active blocks at all times. A larger HCT will have a greater number of unused slots than a smaller HCT containing the same compiled blocks, increasing the hot spot detector sensitivity and encouraging less frequently executed hot spots to be compiled to fill the table. This can be seen in figure 3.11; note that the HCT size axis scale is logarithmic: each doubling of the HCT size only results in an approximately linear improvement in the number of instructions executed in hardware. These diminishing returns are caused by additional space in larger HCTs being filled with less frequently executed hot spots. The levelling off of the graphs for some of the benchmarks occurs when all repeatedly executed blocks have been compiled. The remaining small percentage of instructions that execute in software are either contained within blocks that execute too infrequently to be detected as hot spots even with an extremely high sensitivity (for example blocks that execute only once) or contained within the first few executions of a block that cause it to be detected as a hot spot.

Figure 3.12 shows the number of compilations that take place during the execution of the benchmark with different HCT sizes. The number of compilations generally increases as the HCT size increases, as more blocks are compiled to fill the larger table. In some cases a larger HCT decreases the number of compilations; this occurs when the smaller HCT causes compiled blocks to be displaced (and later re-compiled) as the table is not large enough to hold all currently active blocks. However, the dynamic sensitivity mechanism in the hot spot detector successfully prevents excessive thrashing, keeping the number of compilations with small HCT sizes low.

These results suggest a hardware configuration table size of 32 or 64 provides an acceptable trade-off between the proportion of the program

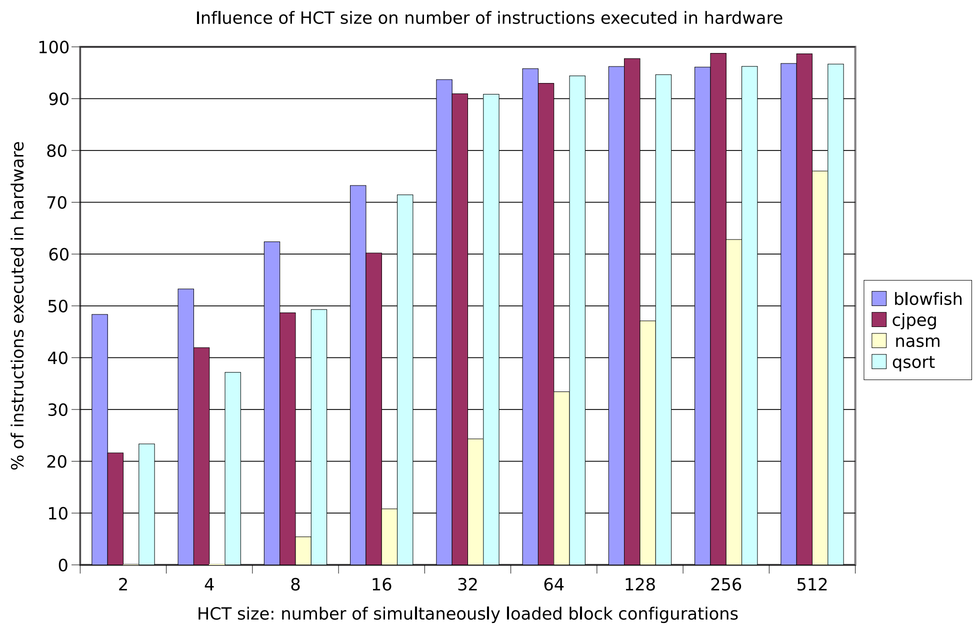


Figure 3.11: Influence of Hardware Configuration Table size on number of instructions executed in hardware

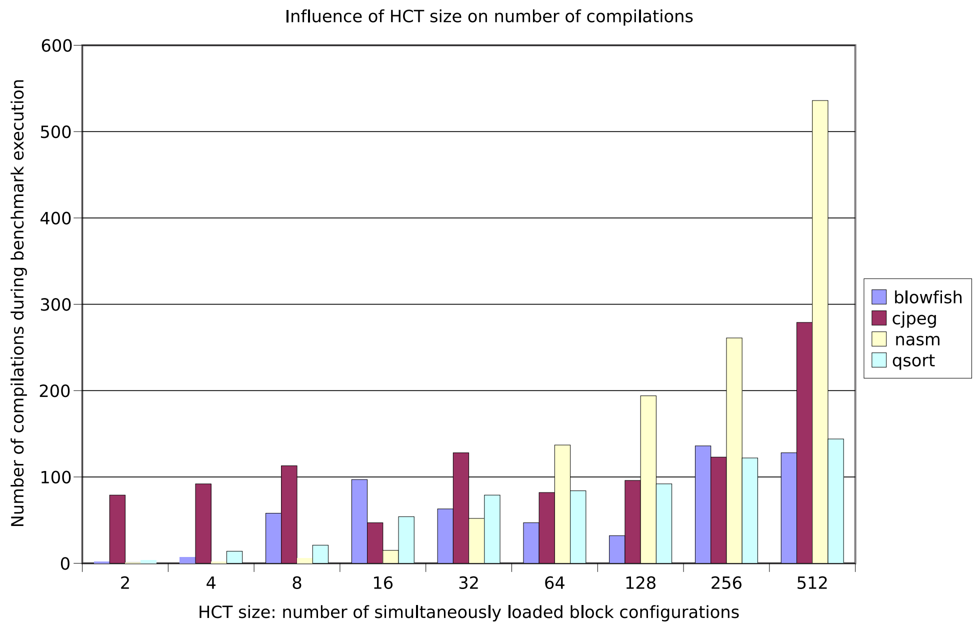


Figure 3.12: Influence of Hardware Configuration Table size on number of compilations

executed in hardware, the number of compilations performed and the size of the HCT itself for the benchmarks used in this analysis.

3.7 Summary

This chapter described methods of profiling code to detect hot spots to be compiled into hardware. The advantages of dynamic profiling over static profiling were discussed and a mechanism to perform dynamic hot spot detection was described. This mechanism consists of a Block Profile Table which monitors the most frequent basic blocks in the currently executing code. Once hot spots are detected they are compiled and stored in a Hardware Configuration Table, from where they can be loaded into the hardware execution engine as required. A table containing 32-64 entries was determined to be a suitable size for both the BPT and the HCT for all of the benchmarks except *nasm*: with this benchmark performance and compilation cost continued to increase with larger HCT sizes. A method of dynamically controlling the sensitivity of the hot spot detector based upon the number of active compiled hot spots was described and analysed; this allows the hot spot detector to adapt to the current program behaviour.

The next chapter outlines optimisations that can be performed on detected hot spots and describes the process of compilation.

Chapter 4

Block optimisation and compilation

This chapter describes the optimisations that can be performed when compiling blocks into reconfigurable hardware. Examples of these optimisations are demonstrated first on single basic blocks and then on chains of blocks to demonstrate advantages of compiling chains into hardware. Section 4.3 describes the process of compilation in more detail.

4.1 Optimisations on single basic blocks

This section describes optimisations that can be performed when compiling single basic blocks.

4.1.1 Temporary values

One type of optimisation that can be performed when compiling a block to execute in reconfigurable hardware is the removal of the need to write temporary values into the register bank. The following basic block is a detected hot block from the *cjpeg* benchmark.

```
1  00011158 MOV      R2, R5 ASR #10
2  0001115C LDR      R3, [R8]
3  00011160 AND      R6, R2, #FF
4  00011164 STRB     R6, [R3], +#1
5  00011168 LDR      R2, [R8, +#4]
6  0001116C SUB      R2, R2, #1
7  00011170 CMP      R2, #0
8  00011174 MOV      R0, R8
9  00011178 SUB      R4, R4, #8
10 0001117C MOV      R5, R5 LSL #8
11 00011180 STR      R3, [R8]
12 00011184 STR      R2, [R8, +#4]
13 00011188 BEQ      000113F0
```

This sequence of instructions requires 24 register bank accesses (15 reads and 9 writes), excluding the program counter. Some values are temporary as they are generated and then later overwritten within the basic block. For example the first instruction writes to R2 which is later overwritten by a load (instruction 5). The value generated by the first instruction is therefore local to this block.

In a load-store architecture such as ARM these temporary values between operations are stored in the register bank. In a reconfigurable architecture these values can be passed directly from the result of one operation to the input of another (via an intermediate buffer). This allows multiple operations to execute in parallel without the need for a complex multi-ported register bank. This can also reduce power consumption: reading from and writing to a simple buffer requires less energy than accessing a large register bank as less or no address decoding is required and the wires can be shorter [HM00].

Only certain hot blocks within a program will be executed in reconfigurable hardware due to compilation overhead and limited hardware resources; the rest will still be executed in software. Values valid at the start and end of the block need to be transferred between the register bank and the reconfigurable hardware. The compilation process needs to identify the registers that are required at the start of the block and ensure their values are passed from registers to the reconfigurable hardware. The registers that are written to during the block need to have their final values updated in the register bank. Other values are temporary within the block: for example a register that is written to before it is read does not need to have its value read from the register bank as it will be overwritten. The following example demonstrates the temporary values within the block:

```
00011158 MOV      R2_1, R5_1 ASR #10
0001115C LDR      R3_1, [R8_1]
00011160 AND      R6_1, R2_1, #FF
00011164 STRB     R6_1, [R3_1 -> R3_2[1]], #1
00011168 LDR      R2_2, [R8_1, #4]
0001116C SUB      R2_3, R2_2, #1
00011170 CMP      R2_3, #0
00011174 MOV      R0_1, R8_1
00011178 SUB      R4_2, R4_1, #8
0001117C MOV      R5_2, R5_1 LSL #8
00011180 STR      R3_2, [R8_1]
00011184 STR      R2_3, [R8_1, #4]
00011188 BEQ      000113F0
```

Values $R4_1$, $R5_1$ and $R8_1$ are valid at the start of the block; therefore, only the values in registers R4, R5 and R8 need to be passed to the reconfigurable hardware at the start of the block. Values $R0_1$, $R2_3$, $R3_2$, $R4_2$, $R5_2$ and $R6_1$ are valid at the end of the block causing R0, R2, R3, R4, R5 and R6 to require updating in the register bank. Values $R2_1$, $R2_2$ and $R3_1$ are temporary values within the block and can be passed directly from the output of the generating operation to the inputs of the operations that require them, bypassing the register bank. The total number of register bank accesses required to execute the block is reduced to nine: three reads and six writes.

[1]The value in R3 changes at this point as this is a post-indexing instruction which increments the base register by 1

A possible method of reducing the register writes further would be to search all possible control paths following the block to determine which values are required by future instructions and which are stored in registers that are overwritten. If a value is required by *any* possible code path it must be written back; if it is overwritten in all paths then it may be discarded at the end of the block. Tracing all possible paths following a basic block may be impractical as the number of possible paths can increase rapidly; additionally indirect branches may not be statically determinable making tracing further paths impossible. If this occurred the search would have to be abandoned and any registers not yet determined must be assumed to be live.

This search would be performed at compile-time, increasing the amount of work performed when compiling a block. In particular, many more instructions must be fetched and decoded during each compilation. Instructions from paths that may never be executed would have to be fetched, possibly polluting the instruction cache. Therefore, this particular optimisation is not performed.

4.1.2 Increasing parallelism

One of the main benefits of compiling to reconfigurable hardware is the ability to increase parallelism over a sequential microprocessor. A data flow graph (DFG) from the example block in section 4.1.1 is shown in Figure 4.1. This shows the maximum parallelism available assuming unlimited execution width in the reconfigurable execution engine, an unlimited number of simultaneous memory and register accesses and a memory load latency of a single cycle. In this case this block of thirteen ARM instructions could be completed in five execution cycles, which is the length of the critical path through the block from the address calculation R8 + #4 to the branch condition test. The maximum number of operations that can execute in parallel is five in the first cycle, but this decreases as the block progresses. By scheduling the

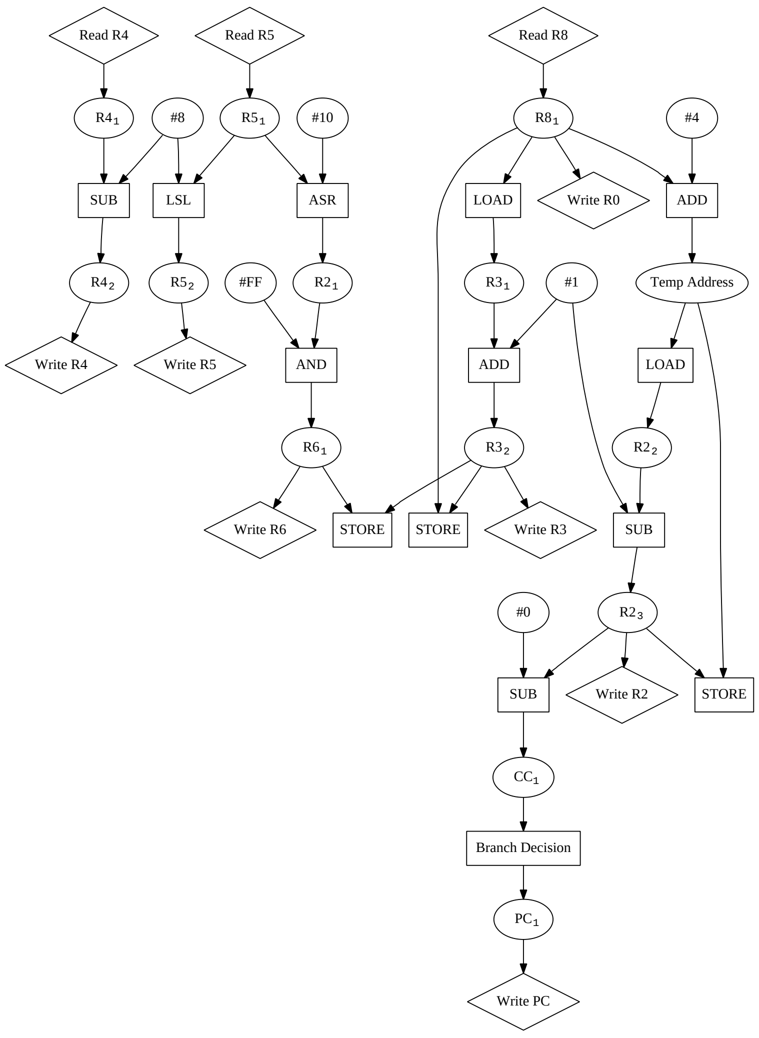


Figure 4.1: Data flow graph of single basic block

subtract and shift operations that write to R4 and R5 respectively to execute later in the block the maximum parallelism required for the block to still complete in five cycles can be reduced to three.

Figure 4.2 shows a DFG for the same block with some constraints imposed. Only a single memory operation is permitted per cycle and a delay of one cycle is introduced following a load to allow for cache latency. The block now completes in six cycles: the additional load delay increases the length of the critical path. The optimum execution parallelism required to complete in this time is three. Reducing the number of operations that can execute in parallel

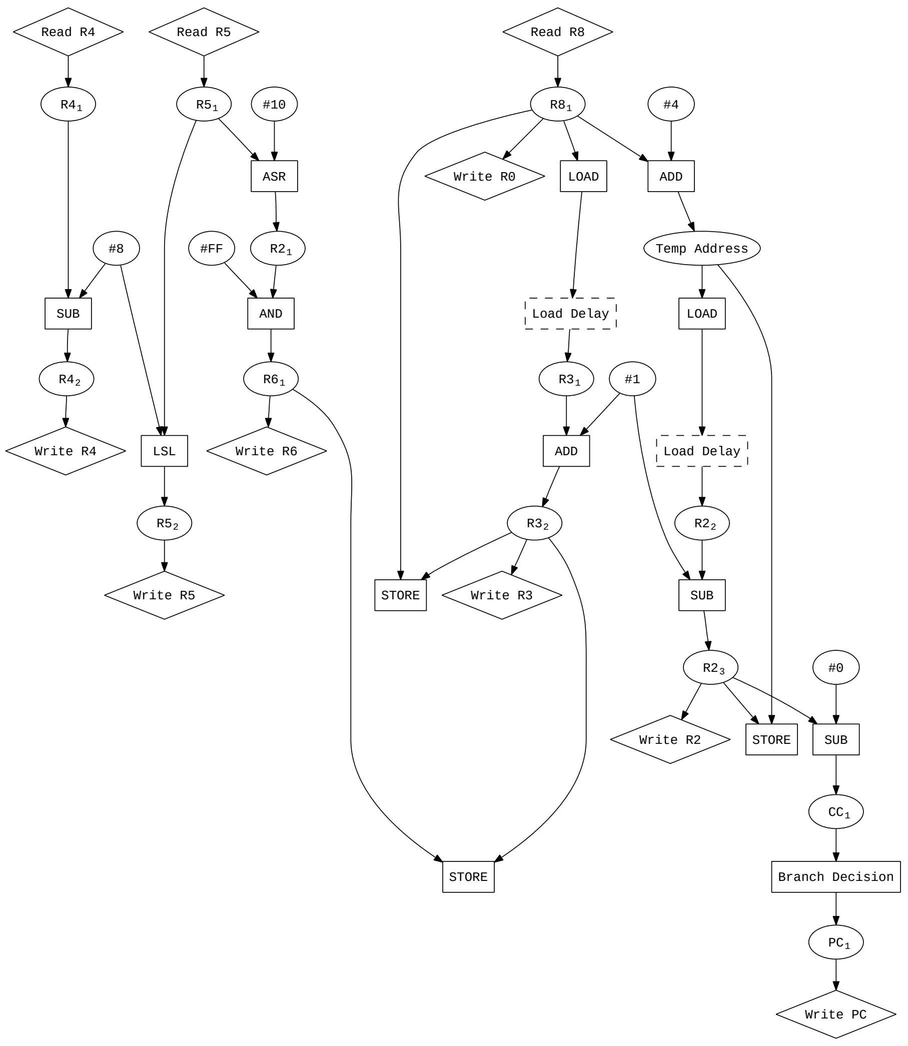


Figure 4.2: Data flow graph of basic block with three-way constrained parallelism, single load per cycle and a single-cycle load delay

to two would increase execution time to seven cycles; increasing parallelism to four will give no additional benefit as the completion time is limited to six cycles by the critical path.

4.2 Optimisations on chains of basic blocks

Section 3.2 discussed how basic blocks linked by highly biased branches can be linked together to form a chain of basic blocks, removing internal control instructions and producing a linear stream of instructions. The fragments of code passed from the hot spot detector to the block compiler are therefore larger than single basic blocks. A key advantage of this is to increase the window upon which the optimisations in section 4.1 are performed. The example on page 77 is the basic block from section 4.1.1 combined into a chain with following basic blocks that are linked with highly biased branches. This chain is taken from the *cjpeg* benchmark and produced using the chaining process described in the previous chapter.

This chain is constructed from five basic blocks; these are shown with different coloured instruction numbers. The conditional branches at the end of each of the first four basic blocks have been determined to be highly biased by the chain builder mechanism discussed in section 3.2. Due to this highly biased nature these branches can be removed and replaced with operations that trigger an exception if the branch causes the chain to exit. This allows the chained basic blocks to become a single large block of instructions with one entry point, one exit point and a linear flow of control.

4.2.1 Increased number of temporary values

The increase in the size of blocks caused by chaining them together increases the number of temporary variables that need not be written into the register bank. In the example in section 4.1.1 three registers were read at the start of

```
00011158 MOV      R2_1, R5_1 ASR #10
0001115C LDR      R3_1, [R8_1]
00011160 AND      R6_1, R2_1, #FF
00011164 STRB     R6_1, [R3_1 -> R3_2], +#1
00011168 LDR      R2_2, [R8_1, +#4]
0001116C SUB      R2_3, R2_2, #1
00011170 CMP      R2_3, #0
00011174 MOV      R0_1, R8_1
00011178 SUB      R4_1, #8
0001117C MOV      R5_2, R5_1 LSL #8
00011180 STR      R3_2, [R8_1]
00011184 STR      R2_3, [R8_1, +#4]
00011188 BEQ      000113F0
0001118C CMP      R6_1, #FF
00011190 BEQ      000113B4
00011194 CMP      R4_2, #7
00011198 BGT      00011158
0001119C CMP      R10_1, #0
000111A0 STR      R5_2, [R8_1, +#8]
000111A4 STR      R4_2, [R8_1, +#C]
000111A8 LDR      R5_3, [R11_1, -#68]
000111AC MOV      R6_2, #1
000111B0 BEQ      0001147C
000111B4 MOV      R3_3, R6_2 LSL R10_1
000111B8 ADD      R4_3, R4_2, R10_1
000111BC SUB      R3_4, R3_3, #1
000111C0 LDR      R2_4, [R8_1, +#8]
000111C4 AND      R5_4, R5_3, R3_4
000111C8 CMP      R4_3, #7
000111CC RSB      R3_5, R4_3, #18
000111D0 ORR      R5_5, R2_4, R5_4 LSL R3_5
000111D4 BLE      0001121C
```

the block and six needed to be written at the end of the block, reducing the number of register bank accesses from 24 to 9. By chaining blocks together the values at the end of a block may be overwritten by the following block due to register re-use, removing the need to write those values back into the register bank. In this example four values ($R2_3$, $R3_2$, $R4_2$ and $R5_2$) that were valid at the end of the single basic block in section 4.1.1 are overwritten by instructions in subsequent blocks. Registers R10 and R11 are read by these added blocks; however, no additional registers are written. Therefore, five registers must be passed to the reconfigurable hardware and six must be written back, giving a total of eleven register bank accesses. This is a significant reduction over the 53 register accesses that occur when the instructions in the chain of blocks are executed in the microprocessor pipeline.

4.2.2 Increase parallelism

Figures 4.3 and 4.4 show data flow graphs for the chained basic blocks on page 77. Nodes labelled as *breakout points* on the DFG correspond to the conditional branches at the end of the basic blocks in the chain. Each of these follows an operation that generates a condition (corresponding to a compare instruction in each of these cases). If the condition is set in a way that would cause the chain to be branched out of then the block must be terminated at this point. Therefore, the *Breakout Points* on the DFG represent the points at which the block may be terminated.

This speculation that the chain of blocks will complete fully allows the control dependencies between the basic blocks to be removed. Figure 4.3 shows the data flow graph for the chained block with control dependencies between each basic block: operations from a later basic block in the chain do not begin executing until the breakout point for the previous basic block has been reached (the colours represent the basic block that each operation is taken

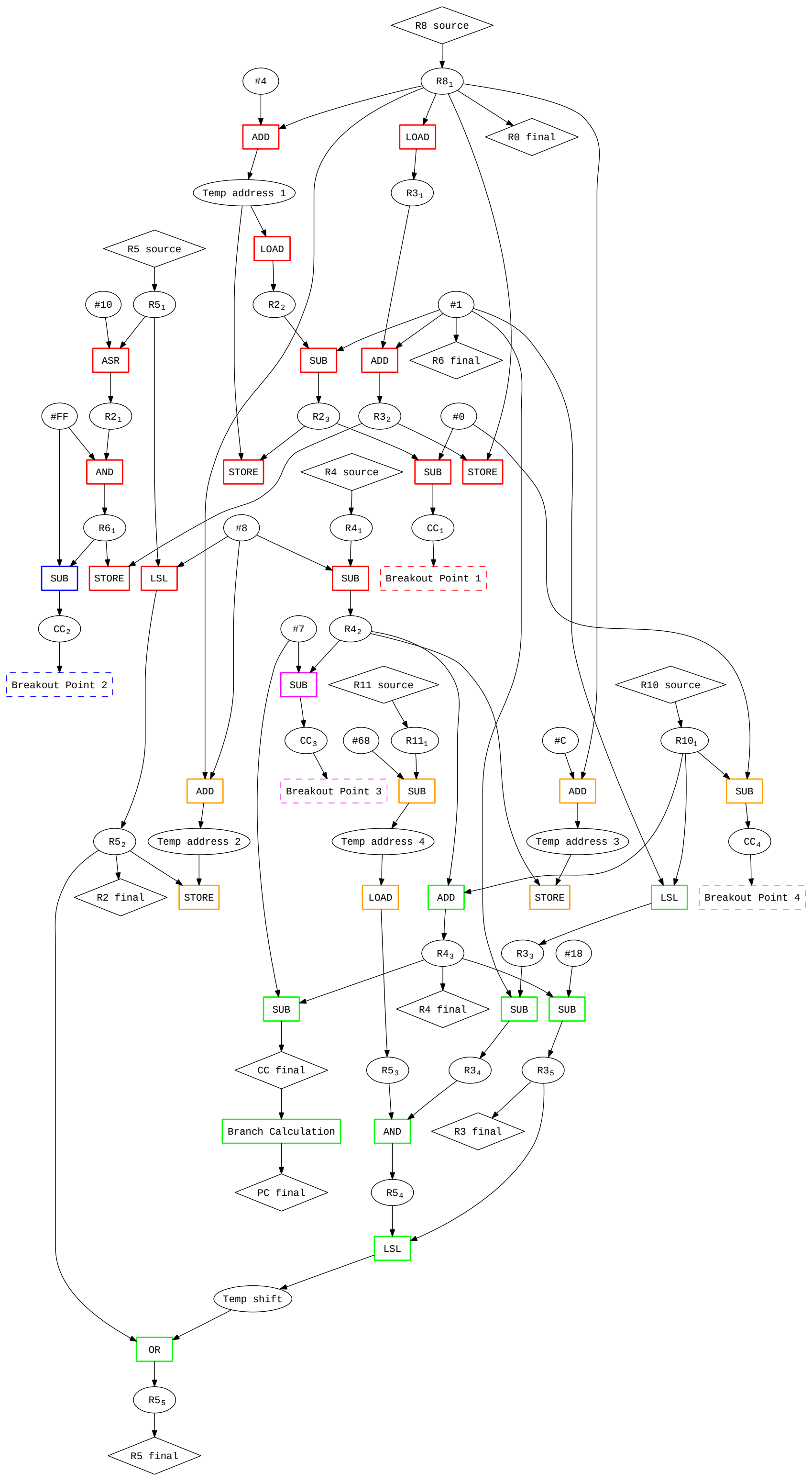


Figure 4.3: Data flow graph of chained block with control dependencies between basic blocks

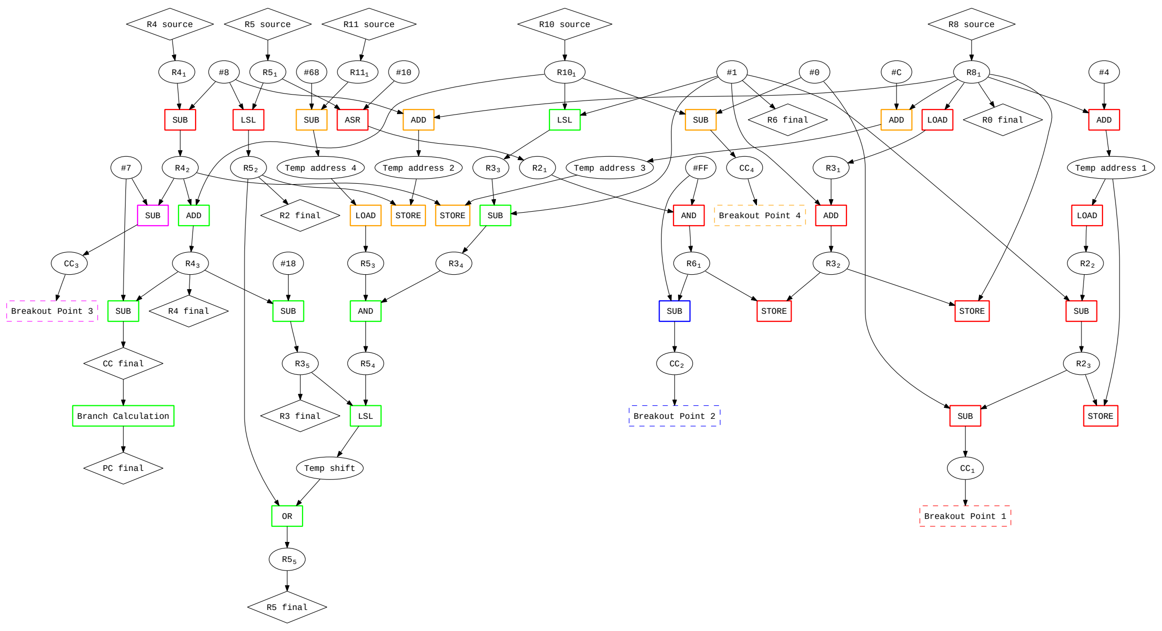


Figure 4.4: Data flow graph of chained block with control dependencies removed

from and correspond to the colours on page 77). In contrast, figure 4.4 shows the same chained block with these control dependencies removed, where operations in later basic blocks may begin executing before the conditional test in the preceding block has completed. This allows parallelism within the chained block to increase significantly. The DFG with control dependencies completes in twelve execution cycles; removing the control dependencies allows the block to complete in five cycles, more than doubling performance, although this assumes an ideal situation with unlimited execution units, unlimited memory bandwidth and single cycle memory latency.

A scheduling of this DFG with a restriction of four parallel operations per cycle is shown in figure 4.5. The scheduling of operations in this DFG is also restricted by only allowing a single load operation per cycle and by adding a single cycle load delay. Here the block can complete in eight execution cycles.

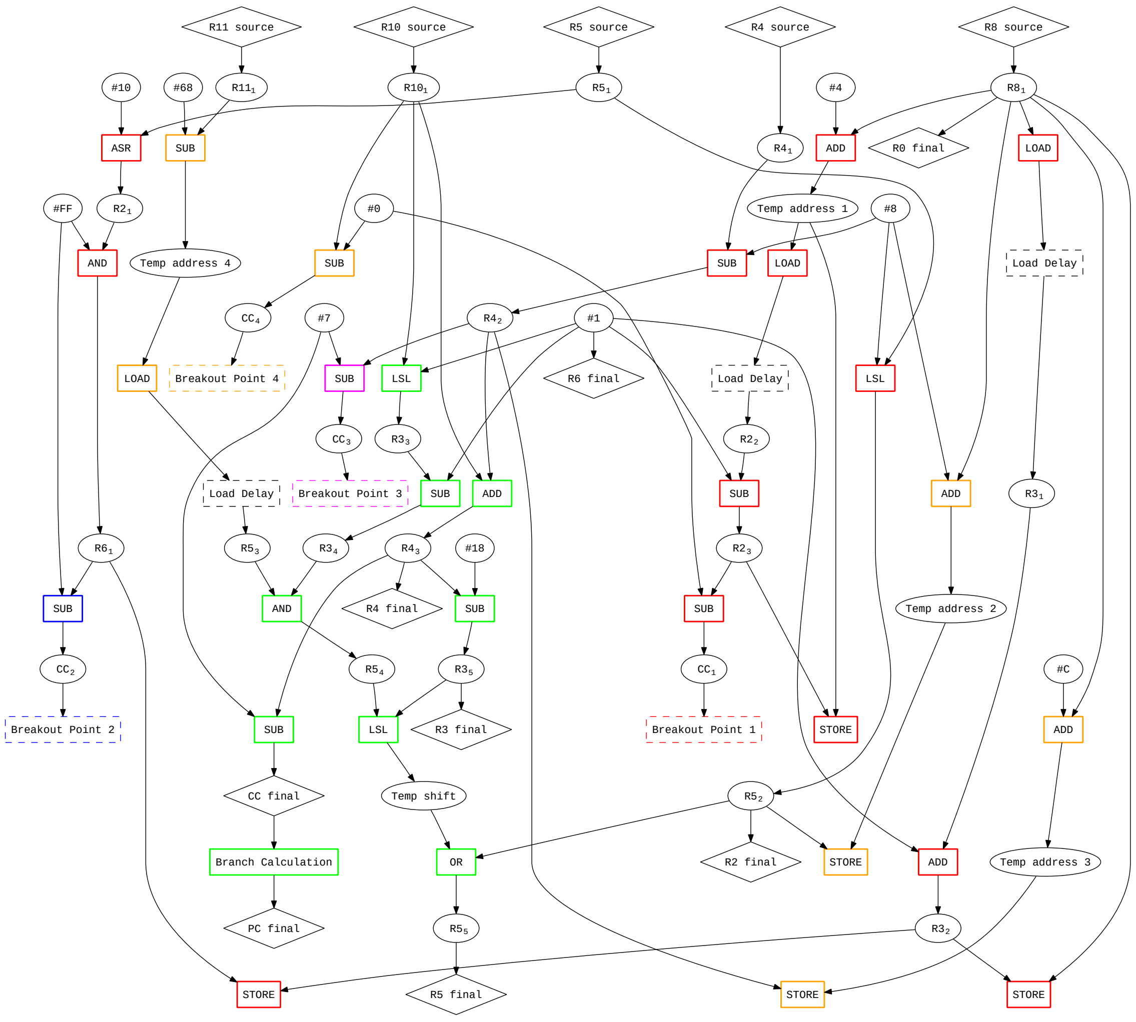


Figure 4.5: Data flow graph of chained block with four-way constrained parallelism, single load operation per cycle and a single-cycle load delay

4.2.3 Memory aliasing

In the above example operations can be scheduled in a different order to how they appear in the instruction stream, according to data dependencies via registers. Loads and stores are scheduled in order only if it can be determined at block compile time that a load uses an address that was previously stored to. In this case the load need not occur as the value to be stored can be passed directly to the operations that require the loaded value. The store must still take place as the value in memory may be required by later instructions after the chain of blocks has exited.

All other loads and stores are permitted to execute out of order. This can cause memory hazards if instructions that alias to the same memory address are moved out of order. If this occurs it must be detected at run-time. By assigning each memory operation a *sequence number*, corresponding to the order of operations in the original code, hardware can be added to detect and handle memory aliasing. A write buffer [MTL95] is used to ensure stores are written to memory in the correct order. This is also used to detect write-after-read hazards caused by a load being moved after a store to the same address; if this occurs the load must fetch the earlier value from memory (or from an earlier store to the write buffer) instead of the later store to the write buffer. Read-after-write hazards can also occur if a load is moved before an aliasing store. In this case the loaded value will be incorrect as the stored value should have been loaded. This is detected by maintaining a table of completed loads (with their sequence numbers). If a write occurs to an address that has been loaded previously from a load with a later sequence number than the write, then a read-after-write hazard has occurred; at this point the block is terminated and re-executed in software. This incurs a cost, but this is deemed acceptable as aliasing loads and stores are very infrequent, as shown in table 4.1. The write buffer and load table are described further in section 5.3.1.

Benchmark	blowfish	cjpeg	nasm	qsort
Chain executions in hardware	833977	9660691	945700	2868825
Block breakouts due to aliasing	0	0	76	0

Table 4.1: Frequency of occurrence of read-after-write memory hazards that are not statically determinable

4.2.4 Reducing stack operations

The ARM Procedure Call Standard (APCS) [Ear03] defines which registers a procedure call must preserve and which may be corrupted. If the procedure requires the use of registers that must be preserved then these registers are pushed onto the stack at the start of the procedure and restored at the end. The following is an example of the typical structure of the stack operations surrounding a procedure call:

```
BL       ADDR
STMDB    R13!, {R4-R7, R14}
...
LDMIA    R13!, {R4-R7, R15}
```

A compiled block containing multiple basic blocks may entirely contain a procedure call and return and this procedure will be effectively inlined into the block. Blocks executing in hardware do not use the architectural registers for holding temporary values within the block; architectural registers are only written to with the final values at the end of the block. Therefore, in cases where a procedure is entirely contained within a block, stack operations to preserve register values are no longer required. This reduces the number of memory operations required to execute the block, improving performance and also potentially reducing power consumption. Table 4.2 shows the proportion of executed hardware blocks containing procedures with stack push/pop operations, and the proportion of stores within executed blocks that are pushes to the stack. This varies significantly between benchmarks: *blowfish* contains

Benchmark	blowfish	cjpeg	nasm	qsort
% of blocks containing procedures	64.3	5.4	1.0	1.8
% of stores procedure call stack pushes	58.0	18.2	5.6	5.6

Table 4.2: Temporary stores to the stack within a block as a proportion of total stores (in executed hardware blocks)

small, frequently executed procedures that can be contained within a hardware block; these are less frequent in *nasm* and *qsort* where frequently executed procedures are not completely contained within hardware blocks due to less-biased paths within the procedure, causing them to not be compiled as single block chains.

However, by removing the stack push operations the external effects of executing in hardware will not be the same as the effects of executing the original software: the stores to the stack that occur when the block is executed in software will not occur when the block is executed in hardware. This may not be a problem as these stores are above the top of the stack after the procedure completes, so in a conventional stack model they are in an undefined area of memory. Programs that use the stack in a conventional fashion and never access memory above the stack pointer will continue to work as expected, however any programs that perform unorthodox stack accesses above the stack pointer may fail if this technique is used.

In addition this method determines which stores can be removed by assuming a standard stack model is being used: for example in the APCS R13 is used as the stack pointer and so the optimisation process would look for stack-like operations using R13 as the base register and remove them if a matching pair was found. However, the registers in the ARM microprocessor are all general purpose and so any register could be used as the stack pointer if the APCS were not adhered to. This could produce incorrect results if R13 was not being used as a stack pointer but appeared, to the block compiler, to be performing stack-like operations within a compiled block.

Not carrying out the pop operations in procedure returns is safe, as the load operation causes no external state change.

The use of this optimisation is potentially dangerous: it should only be used if it is known that a program strictly adheres to procedure call standards and stack models. This optimisation could be optional: it would be disabled by default, but could be enabled by passing an option to the compiler if the executing program is known to behave in a compliant fashion.

4.3 Block compilation process

The block compilation process must transform the instructions output by the hot spot detector into a configuration bitfile usable by the reconfigurable hardware structure. This involves decoding the instructions, building a data flow graph and scheduling operations into a hardware structure with limited parallelism. These are described below using the following sequence of instructions as an example block. Profiling indicates that the return is frequently taken, creating a chain containing two basic blocks.

```
00013CA0 MOV      R12, R13
00013CA4 STMDB    R13!, {R11, R14}
00013CA8 LDR      R3, [R0, #4]
00013CAC SUB      R3, R3, #1
00013CB0 CMP      R3, #0
00013CB4 STR      R3, [R0, #4]
00013CB8 LDRGE    R3, [R0]
00013CBC MOV      R2, R0
00013CC0 LDRGEB   R0, [R3], #1
00013CC4 SUB      R11, R12, #4
00013CC8 STRGE    R3, [R2]
00013CCC LDMGEIA  R13!, {R11, R15}
00009E64 MOV      R2, R0
00009E68 CMP      R2, #0
00009E6C BNE
```

Decode instructions into operations

Firstly each instruction opcode must be decoded into a specification for the operations contained in the instruction. *Operations* are the constituent parts of an instruction, taking one or two operands and performing a single, simple calculation or process (such as a load). Some instructions contain multiple operations: for example a load instruction can contain an address calculation and the load operation itself, ARM data processing instructions may contain a shift operation in addition to the main data processing operation, and ARM store and load multiple instructions consist of many load or store operations.

Each operation is decoded into a specification, which contains the type of operation performed (for example addition, shift, load), location of input operands (register number or immediate value), location to write any result to and whether the operation is conditional. Table 4.3 lists the operations in the example block in execution order.

Remove control operations

Internal control operations (in the form of branches, returns or other instructions that write to the program counter) are removed from the compiled block as it is treated as a single linear block of operations. If a control operation is conditional a *breakout test* operation must be inserted in its place, which causes the block to be terminated if the correct condition is not met.

Non-branching conditional operations generally remain unaltered: the reconfigurable hardware is designed to execute operations conditionally. However, in some cases the condition of an operation will match that of a removed branch. The result of the conditional test is effectively known at compile-time; therefore, the condition on these operations can be set to 'always' or 'never' depending on whether the matching branch is taken or not.

Instr	Operation	Dest	Operand 1	Operand 2	Write CC	Condition
1	Move	R12	R13			
2	Subtract	R13	R13	#4		
	Store		R13	R14		
	Subtract	R13	R13	#4		
	Store		R13	R11		
3	Add	Temp 1	R0	#4		
	Load	R3	Temp 1			
4	Subtract	R3	R3	#1		
5	Subtract		R3	#0	CC	
6	Add	Temp 2	R0	#4		
	Store		Temp 2	R3		
7	Load	R3	R0			GE
8	Move	R2	R0			
9	Add	R3	R3	#1		GE
	Load	R0	R3			GE
10	Subtract	R11	R12	#4		
11	Store		R3	R2		GE
12	Add	R13	R13	#4		GE
	Load	R11	R13			GE
	Add	R13	R13	#4		GE
	Load	R15	R13			GE
13	Move	R2	R0			
14	Subtract		R2	#0	CC	
15	Branch					GE

Table 4.3: List of operations in example block, in execution order

Operations set to ′never′ are removed from the block and operations set to ′always′ are executed unconditionally. The conditional test is still performed for the breakout test of the matching removed branch: if the expected condition is not met the block will be terminated and all results discarded.

In the example block the return in instruction 12 is dependent on a *GE* (signed greater than or equal) condition being generated by the compare, instruction 5. Instructions 7, 9 and 11 also depend on a *GE* condition generated by the same instruction. The profiling performed by the hot spot detector indicates that the conditional return is taken, therefore the *GE* condition is met and so the other instructions with this condition will also execute. The conditional execution can be removed, removing the dependency between the compare and these instructions, increasing parallelism within the block. If, alternatively, the profiling had indicated that the return is not taken then the *GE* condition will return false and so the operations can be removed.

Identify values and remove duplicate operations

Once a list of operations has been generated the *values* within the block must be determined. Values correspond to single assignments to registers; each new write to a register results in a new value being created. Each value generated or read during the block is given a unique value identifier. Table 4.4 shows the operations with individual values labelled with the same notation as the example on page 72. In addition to values in registers, value identifiers are assigned to each unique immediate value used (in this example 4, 1 and 0) and for temporary values not stored in a register in the original code, such as calculated addresses within load or store operations.

Move operations merely duplicate values and so can be removed, with the value identifier of the move destination value renamed to that of the source value. In the example the first instruction moves R13 to R12; subsequent uses of R12 (in this case the subtract in instruction 10) are renamed to the initial value in R13, $R13_1$.

Operations that have the same operand value identifiers, the same operation type and the same condition are duplicates and can be removed. In the example the address calculation for the load and store in instructions 3 and 6 both add R0 to #4. R0 does not change between these operations therefore they are duplicates. Subsequent uses of the result from the second of these address calculations (*Temp 2*) are renamed to take the value from the first (*Temp 1*) and this second operation is removed from the list.

Assigning unique identifiers to each value makes dependencies between operations explicit rather than implied by instruction order and storage location. This allows a dependency graph to be constructed.

Identify memory aliases

It may be possible at compile-time to determine that two memory operations access the same location in memory. Memory accesses that use the same value will point to the same memory address. This information is used to prevent statically determinable read-after-write memory hazards by ensuring loads with addresses that match earlier stores are not scheduled out of order. It can also be used to remove loads that follow a store to the same address: the value can simply be forwarded directly.

In this example both a load and store operation take place at the address in *Temp 1*, however this is a store following a load and so the load cannot be removed. A write-after-read memory hazard will not occur in this case as the store depends on the load and so they will not be scheduled out of order.

Furthermore, there is a matching pair of push and pop operations to the stack. If a conventional stack model is assumed these operations can be removed, as was described in section 4.2.4.

Operation	Dest	Operand 1	Operand 2	Write CC	Condition
Subtract	$R13_2$	$R13_1$	Imm 1		
Store		$R13_2$	$R14_1$		
Subtract	$R13_3$	$R13_2$	Imm 1		
Store		$R13_3$	$R11_1$		
Add	Temp 1	$R0_1$	Imm 1		
Load	$R3_1$	Temp 1			
Subtract	$R3_2$	$R3_1$	Imm 2		
Subtract		$R3_2$	Imm 3	CC_1	
Breakout					CC_1
Store		Temp 1	$R3_2$		
Load	$R3_3$	$R0_1$			
Add	$R3_4$	$R3_3$	Imm 2		
Load	$R0_2$	$R3_4$			
Subtract	$R11_2$	$R13_1$	Imm 1		
Store		$R3_4$	$R0_1$		
Add	$R13_2$	$R13_3$	Imm 1		
Load	$R11_1$	$R13_2$			
Add	$R13_1$	$R13_2$	Imm 1		
Subtract		$R0_2$	Imm 3	CC_2	
Branch					CC_2

Table 4.4: List of operations in example block with individual values labelled and internal control operations removed

Build dependency graph and remove redundant operations

Once individual values have been identified, a data flow graph of the block can be constructed. The DFG for the example block is shown in figure 4.6. Marked in red on this diagram are operations that do not write a result back to registers or memory, do not generate a condition used by a breakout test and do not have any dependencies that perform either of these. In the example code R11 is written to by a subtract operation which is later overwritten by the stack pop operation in instruction 12. The value written to R11 is therefore never used and so need not be generated, allowing this operation to be removed from the block.

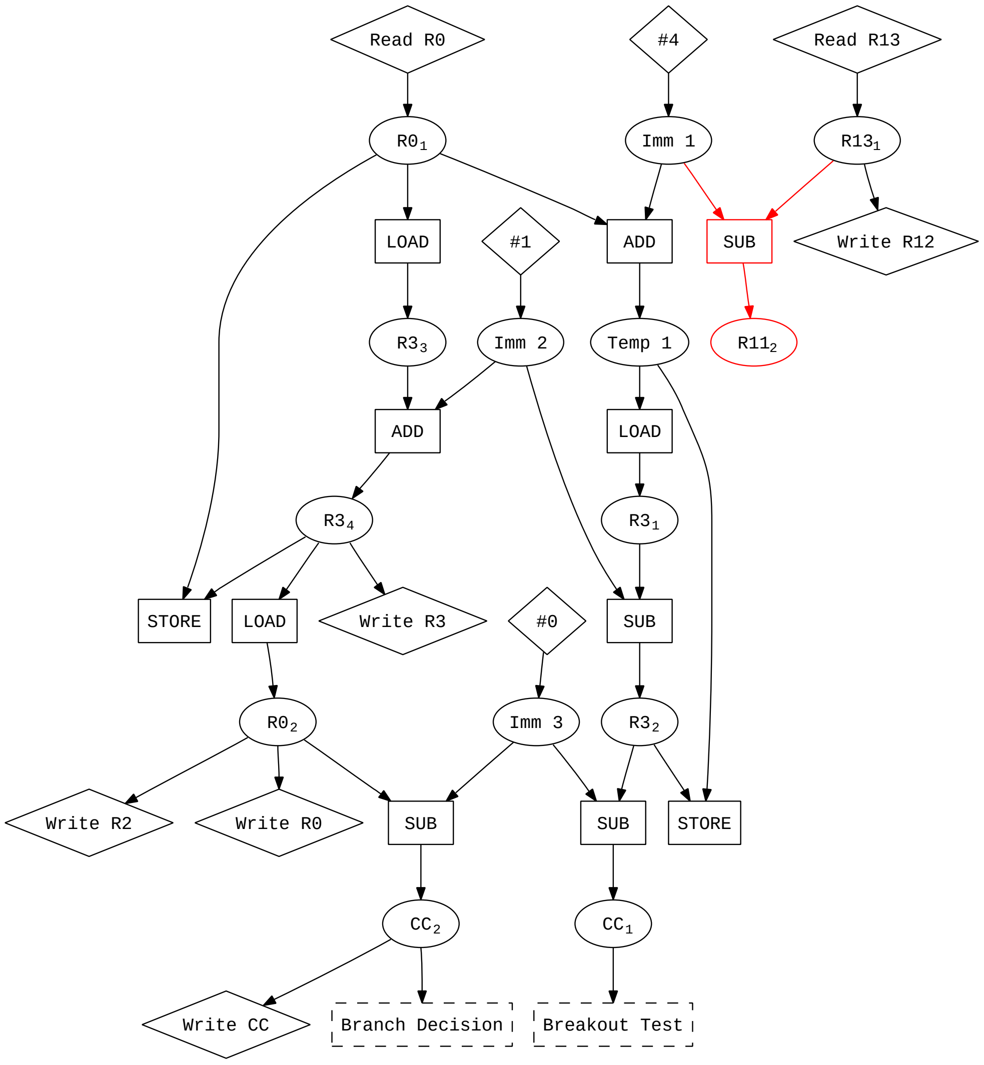


Figure 4.6: Data flow graph of example block, with redundant operations in red

Prioritise operations for scheduling

Once a data flow graph has been constructed, operations must be scheduled within the reconfigurable execution engine.

Many blocks contain a large number of operations that are not dependent on others and so could execute in the first cycle (in a system with unlimited parallelism). The remaining operations depend on one or more of these operations. Some of these operations will have a long chain of dependent operations, others will have no dependencies. This typically leads to a triangular DFG, with many operations that have no or few dependencies and a tail of operations that make up the critical path. This can be seen in figure 4.4: ten operations are not dependent on any others and can execute immediately in the first stage; this decreases in each successive stage until the final stage only contains a single operation.

The hardware execution engine has a limited number of functional units, giving a fixed parallelism in each execution cycle. This may be less than the available parallelism at the start of the block, but greater towards the end. Some operations from the first cycle may therefore be moved to later cycles to allow the block to fit within the hardware. To ensure the block completes within the minimum number of execution cycles, operations with a long path of dependencies or large number of dependent operations should be scheduled as early as possible within the block. Operations with few or no dependent operations can be moved to available slots towards the end of the block.

Additional constraints on scheduling are imposed by the design of the reconfigurable hardware structure proposed in chapter 5. The memory interface is one example: this has a limited bandwidth of one load per cycle. In addition, the latency of the memory interface means that loaded values are not available in the next execution cycle. Load operations are likely to have a number of dependencies within the block, and so they should be prioritised and scheduled as early as possible. Additional detail on the scheduling of operations into the proposed hardware structure is given in section 5.4.

4.4 Performing the compilation process

The processes described in the previous section must be performed at run-time. This could be done in software; however, this would require the currently executing program to be stalled, impacting performance. Another method is to use a dedicated piece of hardware; this would not affect performance as it could run simultaneously with the currently executing program. However, the additional hardware required would increase the area and static power consumption of the system. As the number of compilations is low this hardware would be infrequently utilised, sitting idle once hot spots had been compiled, and would therefore be an inefficient use of resources.

Alternatively, a compilation program could execute within the reconfigurable hardware instead of the microprocessor. This would effectively utilise the reconfigurable hardware as a type of *Instruction Path Coprocessor* (I-COP) [CS00]. An I-COP is a programmable piece of hardware that can be used to transform traces of instructions at run-time, for example to decode CISC instructions into sequences of RISC-like operations, allowing them to be executed by a simple, efficient processor. I-COP programs can also be used to perform optimisations on traces of instructions in a trace cache. I-COP programs have been shown to contain high instruction-level parallelism and perform well in VLIW [CS00] and reconfigurable pipeline architectures [CPSS00].

To allow the reconfigurable hardware to perform the compilation process the execution of hot spots in hardware could be temporarily disabled; execution of the program could continue in software during the compilation process. This would improve performance compared to stalling the entire system and compiling in software. However, the compiler program would contain loops and other control structures and therefore be more complex than the simple sequences of operations normally processed. This would require an increase in the complexity of the sequencer that controls the reconfigurable hardware; however, the additional hardware required would be less than that for a dedicated hardware compiler.

The compilation process must be supplied with a trace of instructions contained within the hot spot. If the compilation process is performed in software these can be fetched directly from memory (the hot spot detector needs only to supply the compiler with a list of the basic blocks in the hot spot). If the compilation process is performed in dedicated hardware or the reconfigurable hardware structure the software will continue to execute in the microprocessor and is likely to execute the hot spot to be compiled in the near future. When this occurs the instruction opcodes fetched in the hot spot can be buffered and forwarded to the compiler, removing the need for the compiler to fetch the instructions itself and so reducing the number of memory cycles required.

Performing the compilation process in the reconfigurable hardware structure itself provides a compromise between performance, hardware utilisation and added hardware complexity.

The mapping of data flow graphs into the reconfigurable hardware structure is described in section 5.4.

4.5 Summary

This chapter describes the optimisations that can be performed by compiling blocks of instructions into hardware. It outlines the process of applying these optimisations to a sequence of code, generated by the hot spot detector, to construct a data flow graph to be used in configuring the hardware.

The improvement to these optimisations gained by constructing larger blocks from chains of basic blocks is also discussed. These include increasing parallelism, reducing register bank and stack accesses, and removing operations no longer required due to a fixed control path being taken through the block.

The next chapter discusses a suitable hardware structure for executing detected hot spots.

Chapter 5

Reconfigurable hardware structure

The previous two chapters described the processes of hot spot detection and block compilation. For each detected hot spot these processes produce a data flow graph with control information (in the form of infrequently executed branches between basic blocks) removed. The intention is for these data flow graphs to be executed by a hardware execution engine that is more energy-efficient than the standard microprocessor execution pipeline. This chapter discusses possible options for this hardware structure and describes a suitable architecture.

5.1 Advantages of executing in reconfigurable hardware

A number of methods can be used in the design of the execution engine to improve performance or energy efficiency compared to a standard microprocessor pipeline.

Increased parallelism

Superscalar microprocessors increase performance over scalar microprocessors by determining dependencies at run-time and allowing independent operations to execute in parallel functional units. However, dependencies must be detected each time instructions are executed, increasing hardware complexity and power consumption significantly.

Very Long Instruction Word (VLIW) architectures [Fis83] execute large instructions that contain multiple parallel operations. Dependency checking and parallelising of operations is performed statically, removing the overhead of performing it at run-time. However, one disadvantage of VLIW architectures is that the number of functional units is fixed in the instruction set: binaries compiled for one generation of processors will not perform better on newer hardware versions where improved technology allows for greater numbers of functional units.

Explicitly Parallel Instruction Computing (EPIC) processors [SR00] provide one solution to this problem. Like VLIW architectures, instructions that can execute in parallel are determined at compile-time; however, unlike VLIW, the scheduling is performed by the processor at run-time. The compiler does not need to know the internal structure of the hardware, therefore allowing greater code compatibility between different generations of hardware.

An alternative is to translate instructions into a parallel representation the first time they are executed. This reduces run-time parallelisation overheads as dependency checking is only performed once for each section of code, but maintains compatibility with existing software. Transmeta's Crusoe architecture is an example, translating x86 instructions at run-time to execute in a VLIW machine [Kla00].

This last method is used in this system to extract parallelism at run-time. Dependencies in detected hot spots are determined at block compile time and operations are scheduled to execute in parallel in the reconfigurable hardware structure.

Reduced register bank accesses

In a standard microprocessor, values are stored in a register bank between instructions. A reconfigurable hardware structure allows temporary values to be forwarded directly between operations via a local register as opposed to a large multi-ported register bank. This allows for increased parallelism by bypassing the register bank, which can be a significant performance bottleneck in high performance microprocessors. Additionally, this may reduce power consumption as writing and reading simple buffer registers requires less energy than accessing a register bank. Register values still need to be transferred between the architectural registers and the hardware structure, however this can be reduced to one read and write for each used and written architectural register respectively, at the start and end of the block. This was discussed in section 4.1.1.

Reduced control overhead

Section 3.2 described the chaining of basic blocks together based on highly biased branches. This allows these branches to be removed (though the branch test must still take place) which removes the need to perform the branch calculation and allows many optimisations to be performed. These include removal of operations whose results are overwritten, reduction in temporary values written to the register bank and increased parallelism due to larger block size and reduced control dependencies. These were described in section 4.1.

However, removing internal branches from the block also has a number of disadvantages. The profiling hardware must monitor branch bias to determine whether the branch can be removed, and as loads and stores may be speculatively re-ordered hardware must be added to detect memory hazards (see section 5.3.1).

5.2 Design aspects of the hardware execution engine

This section discusses aspects that must be considered in the design of the hardware execution engine and discusses possible options. Section 5.3 proposes a suitable structure.

5.2.1 Interface with microprocessor

Reconfigurable execution engines can be sited in different locations within a computer system. These can vary from very tightly coupled functional units within a microprocessor pipeline down to units completely independent of a microprocessor. The coupling of these hardware structures depends on the type of reconfigurable structure and the algorithms being processed. For example streaming algorithms fetch data from memory in very predictable patterns and are highly repetitive; this data can be supplied to a reconfigurable streaming engine by a DMA unit, therefore the interface between the processor and the reconfigurable hardware needs only to be for control purposes. The processor will initialise the hardware (for example with a memory pointer to the start of the data to be processed) and allow it to run. The interaction between the processor and the reconfigurable engine is small and infrequent, so a low latency high bandwidth interface between the two is not important for high performance. Therefore, they can be loosely coupled: for example the reconfigurable hardware could be located off-chip and interface with the main processor via the main system memory.

A reconfigurable engine that provides additional, configurable instructions in a processor requires more frequent interfacing than the streaming processor in the previous example as the instructions will be more frequent and run for a shorter period of time. This is likely to require a low latency interface. Therefore, the reconfigurable hardware will need to be more tightly coupled with the processor than in the streaming processor example. This could be in the form of an on-chip coprocessor or configurable functional units within the processor datapath.

The system presented here is required to execute data flow graphs with a linear flow of control and a size corresponding to a small number of basic blocks of equivalent microprocessor instructions. These blocks execute for a shorter period of time than streaming algorithms, but for longer than single instructions: essentially they can be thought of as very complex customised instructions. As the blocks represent only small portions of the program, they are called very frequently. The compiled blocks use data in architectural registers, therefore values must be able to be transferred between the reconfigurable hardware and the microprocessor's register bank. The short execution time, frequent switching between processor and reconfigurable hardware control and the desire to access architectural registers directly suggests that optimal performance will be achieved with a low-latency, tight coupling between the hardware structure and the processor.

5.2.2 Parallelism

The number of parallel functional units should reflect the available parallelism in the data flow graphs. Reducing the number of functional units will result in a smaller, simpler hardware structure but may limit performance if the available parallelism is greater. Having a greater number of functional units than the available parallelism will give no performance improvement and

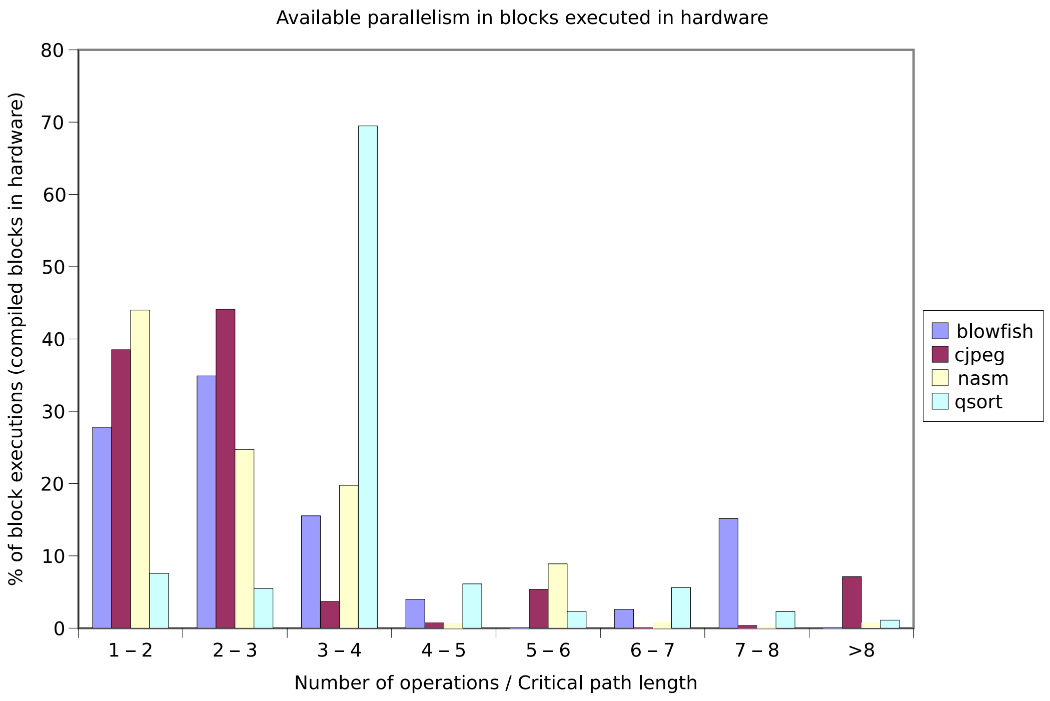


Figure 5.1: Average parallelism in executed hardware blocks

result in functional units being under-utilised. Increasing the number of functional units will increase the size of the hardware engine, increasing its area and static power consumption. Additionally, more configuration data will be required to configure the functional units and the more complex interconnect required, increasing the on-chip storage required. Therefore it is advantageous to have the smallest number of functional units for the available parallelism in the data flow graph to be processed.

Figure 5.1 shows the theoretical maximum average parallelism of detected and compiled hot blocks executed in the hardware structure. The relative execution frequencies of blocks with different parallelism ratios, calculated by dividing the total number of operations in the block by the critical path length, are shown. This approximates to the maximum average parallelism which could ever be exploited in each block. This figure shows that around 65-80% of blocks in the *cjpeg*, *blowfish* and *nasm* benchmarks have a maximum average parallelism of between one and three; however, during the execution of *qsort*

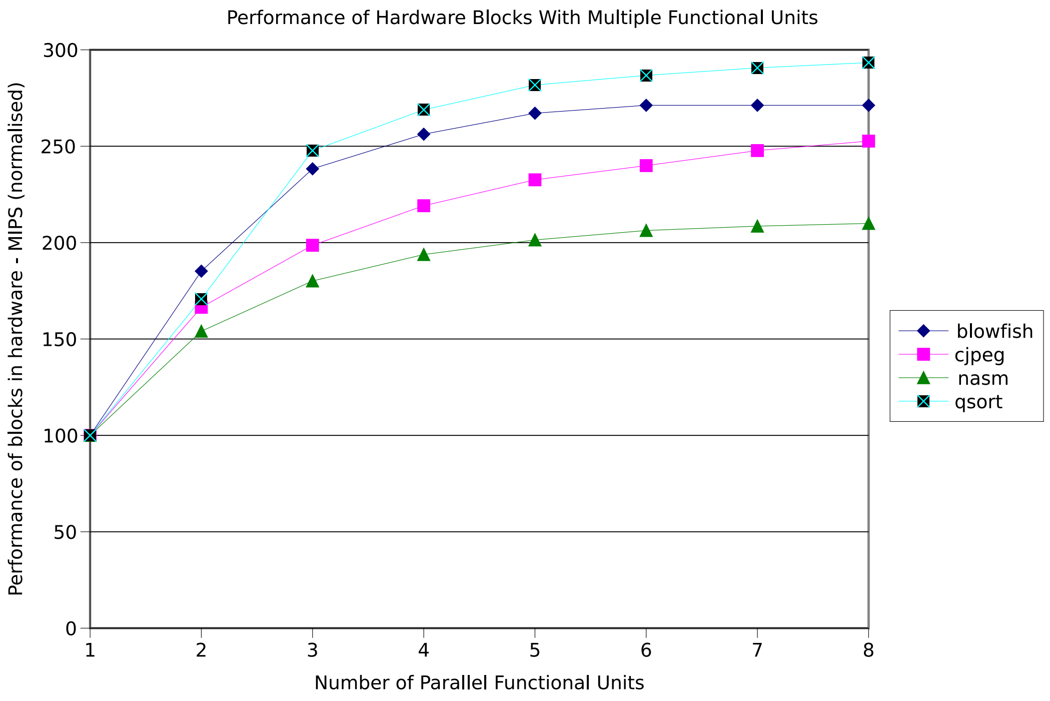


Figure 5.2: Performance of hardware blocks with increasing numbers of parallel functional units

70% of blocks have a parallelism of between three and four, showing a greater potential parallelism available in this benchmark. Few blocks (<20%) in any benchmark have an available parallelism over four. These results suggest that increasing the number of parallel functional units in the hardware structure over four would have little improvement on performance, yet would increase area and static power consumption.

Figure 5.2 shows the total number of cycles required during the benchmark run to execute the detected hot blocks in hardware with different numbers of parallel functional units, as a percentage of the number of cycles required with a single functional unit. Two functional units increases performance significantly over one in all benchmarks, increasing to between 150% and 180%. Further increases in the number of functional units provide smaller improvements, with performance levelling out between 210-290% of the performance of a single functional unit, depending on the benchmark.

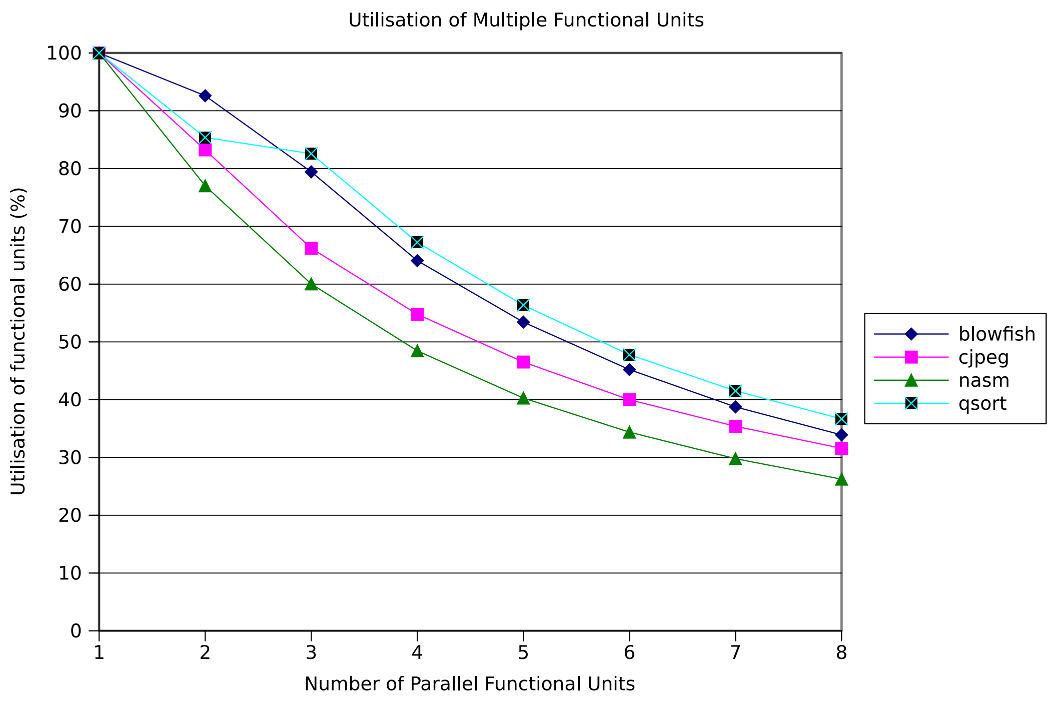


Figure 5.3: Utilisation of hardware structure with multiple parallel functional units

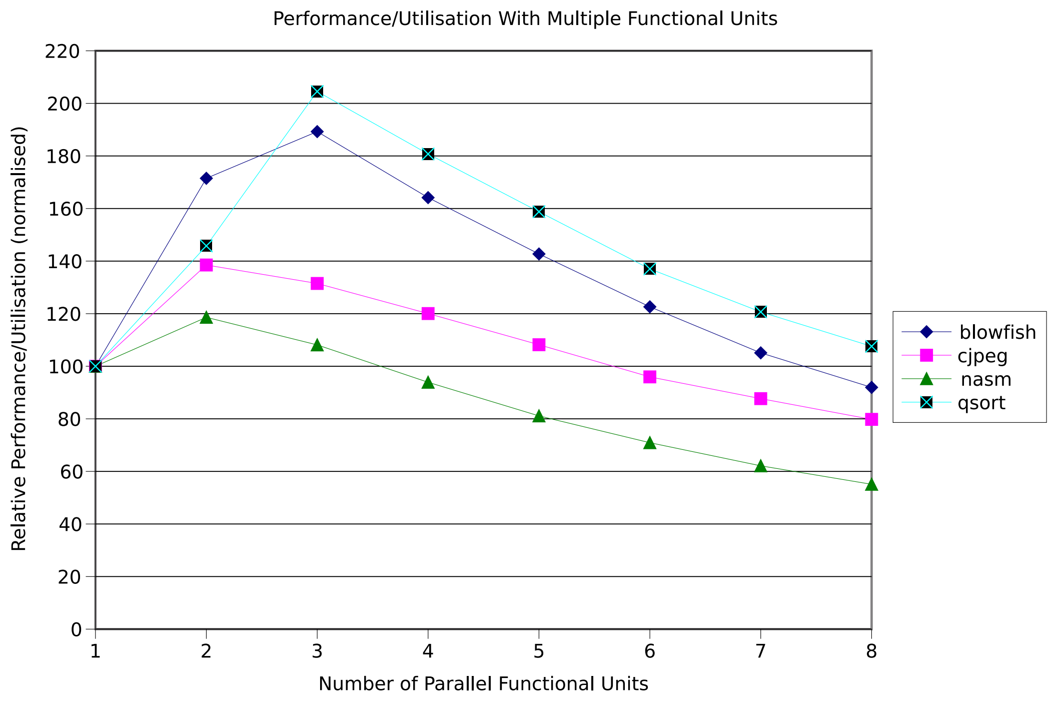


Figure 5.4: Relative performance over utilisation of hardware structure with multiple parallel functional units

The utilisation of functional units with different numbers of parallel units is shown in figure 5.3. This gives an approximation of the efficiency of the system: in an ideal situation with unlimited parallelism in the code the utilisation would remain at 100%. However, figure 5.2 shows that the functional units can not be fully utilised due to limitations in the available parallelism in compiled blocks. Therefore a trade-off must be made between performance increases and decreases in utilisation and its effects on static power consumption and area when determining a suitable number of parallel functional units.

Performance over utilisation is shown in figure 5.4. This gives a bias to performance while taking into account the estimated decrease in utilisation from increasing the number of functional units. Three functional units gives the highest performance/utilisation ratio in *blowfish* and *qsort* as these benchmarks gain a greater performance increase from two to three functional units than *nasm* and *cjpeg*, which have the greatest performance/utilisation with two functional units.

The results assume that configurations can be scheduled efficiently into the available functional units: they do not take into account restrictions due to data routing, transfers to or from memory and registers, or specialised functional units. Increasing the number of functional units improves the flexibility of the hardware structure allowing data flow graphs to be scheduled more easily into it, reducing the complexity of the interconnect required and simplifying the compilation process. More functional units than suggested here may therefore be beneficial.

5.2.3 Memory and register interface

Although the hardware structure operates on values held in its internal registers, it must read values from the architectural registers at the start of the block and write values back into the bank once the block has completed.

In addition, values may be read from or written to memory. Restricting the maximum number of possible parallel memory or register reads and writes reduces the complexity of the system, but may impede performance.

Memory reads

The performance scaling with parallel functional units shown in figure 5.2 assumed an unlimited number of memory reads per cycle. This is not practical in a real implementation as there is a limit on the bandwidth of the memory bus and the number of memory reads at different addresses that can be performed in each cycle. Improved memory bandwidth can be achieved by increasing the width of the memory data bus, but this is only effective if the values to be fetched are in contiguous blocks of addresses in the memory. Increasing the number of simultaneous memory reads at non-contiguous addresses requires multiple read ports in the data cache (or faster memory); this increases hardware complexity and power consumption.

Figure 5.5 shows that a restriction of a single load per cycle does not affect performance significantly. The results were gathered with the assumption that the hardware engine contains four parallel functional units, and used the scheduler described in section 5.4 (with a modified load limit per cycle). This figure shows the increase in the number of hardware execution cycles during the benchmark when the number of loads is limited to one or two per cycle. A limit of two loads per cycle shows no significant performance decrease when compared to an unlimited load bandwidth. When the number of simultaneous loads is limited to one the number of cycles in *qsort* increased by 20% over an unlimited load bandwidth; in the other three benchmarks it increased by 5-8%. As *qsort* is a sorting algorithm it performs fewer data processing operations for each value loaded, explaining why this benchmark requires a larger number of loads per cycle than other benchmarks (that perform more complex data processing) such as *cjpeg*. This decrease in performance

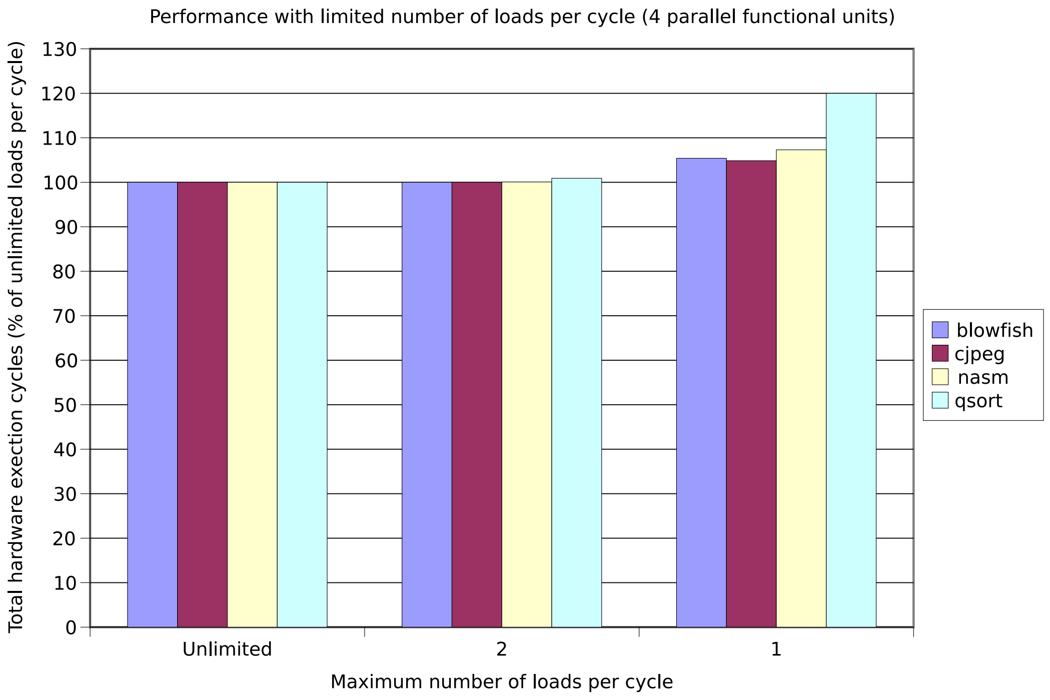


Figure 5.5: Performance of hardware blocks with limited numbers of loads per execution cycle

is deemed acceptable in this case to retain the simplicity and lower power consumption of a standard, single-width memory interface.

Register reads

Register reads are relatively straightforward: the hardware and software execution engines will not be in use simultaneously, so the same register read ports can be used for both. The number of register reads within a block is relatively small as, normally, a register will only have to be read once. After it has been read its value can usually be stored within the hardware structure if it is to be reused (in some cases it may be beneficial to read a value twice to free storage within the hardware: for example where a register value is only used at the start and end of a block). Although the number of registers read throughout a block is low, each functional unit will need to read a value from registers in the first execution cycle of the block.

Memory writes

The chaining together of basic blocks by removing highly biased branches increases the potential for performing optimisations as described in section 3.2. However this means that a hardware block may not complete if the condition test for a removed branch does not return the expected value. At this point the block is abandoned and control returns to the start of the block, with the microprocessor re-executing the block in software. It must therefore be possible to restore architectural and memory state to what it was at the start of the block until it is known that the block will complete fully (once the last point the block can break out from has been passed).

Store operations therefore must not be committed to memory until this point has been reached. The solution to this is to use a *write buffer*. This temporarily holds stored values until after the block completion point, at which point the values in the buffer can be committed to memory. The buffer must be checked during a load operation for matching addresses as a value yet to be stored at that location may be held in the store buffer.

An additional advantage of using a write buffer is that writes from the reconfigurable hardware are not limited by memory bandwidth. Values in the write buffer can be committed to memory when free memory cycles become available (only if the buffer fills up will the processor have to be stalled to free the memory bus allowing it to empty). Allowing multiple simultaneous stores to a write buffer requires significantly less additional hardware complexity than modifying a memory interface to support multiple simultaneous memory accesses. In addition, a write buffer can be written to in parallel with a memory load. Figure 5.6 shows how limiting the number of stores in each cycle affects performance. As with limiting the number of loads, *qsort* is the most significantly affected of the four benchmarks when the number of parallel stores is limited, experiencing an 18% increase in the number of hardware execution cycles. The effect is smaller than that observed by limiting loads

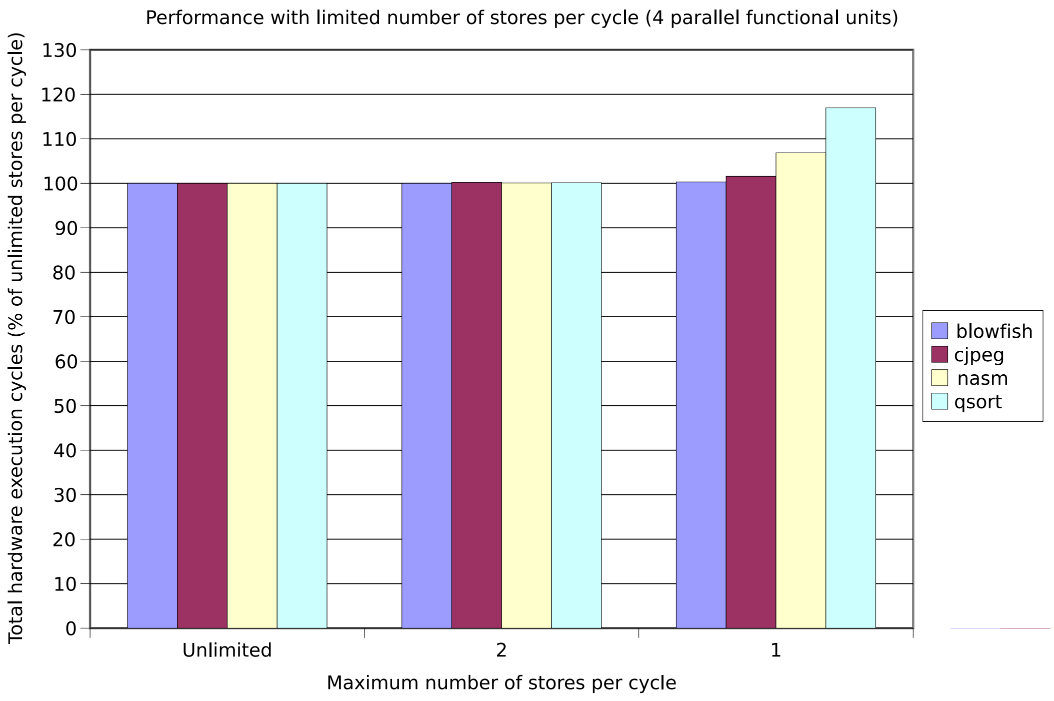


Figure 5.6: Performance of hardware blocks with limited numbers of stores per execution cycle

as many operations can depend on the result of a load, whereas a store has no dependent operations and so can be scheduled at any time, for example in free slots towards the end of a block with a long critical path.

Register writes

Register writes must be handled in a similar fashion to memory stores: they cannot be committed until it is known that the block executing in hardware will complete. One solution would be to use a write buffer similar to the one used for memory stores. Upon block completion the contents of this buffer would have to be written into the register bank. The processor would either have to be stalled during this time, incurring a performance penalty, or the buffer would have to be checked for unwritten register values on each register read, which would require additional hardware and increase power consumption when reading registers.

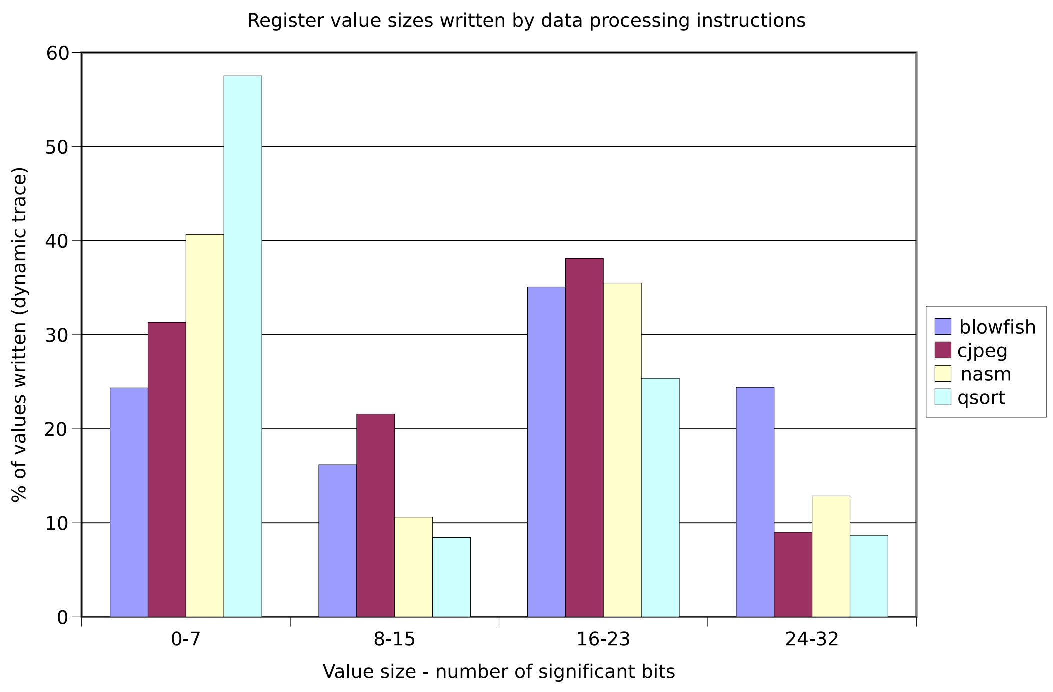


Figure 5.7: Size of values (position of most significant bit) written by data instructions

Another solution is to have two locations for each architectural register and use register renaming to select the one to be used. One location would preserve the original value of the register (at the start of the block); the other would be written to by the block executing in hardware. The register holding the correct value at the end of the block (depending on whether the block completed successfully or was broken out of) would be renamed to the appropriate architectural register and would be used for subsequent execution. This would allow execution to continue immediately at the end of a block.

5.2.4 Word size of functional units

One advantage of reconfigurable hardware over a microprocessor pipeline is its potential to configure functional units to fit the data sizes to be processed. For example, if an algorithm processes 8-bit values a reconfigurable system can be configured with 8-bit functional units, whereas a microprocessor would have to use its fixed width ALU, which may be 32 bits or larger. This improves hardware utilisation and power efficiency. Figure 5.7 shows the size of values during a dynamic trace. These results are only estimates determined by the position of the most significant bit in values written by data processing instructions (such as additions or logical operations). Many values are fewer than eight bits in size, particularly for *qsort* due to this benchmark sorting byte-sized values. A further peak occurs at 16-23 for all benchmarks. This is attributable to the size of addresses in these benchmarks typically falling within this range.

To create a reconfigurable hardware configuration with functional units tailored to the size of the data in the algorithm the size of these values must be known at the time the configuration is created. This is possible when the reconfigurable hardware is programmed by hand, or the configuration is created by a compiler using a programming model that allows the size of data values to be specified.

In the system proposed in this thesis the configuration for the hardware is determined from a compiled binary by performing dynamic profiling. Although it would be possible to profile the size of data values this would increase the cost of profiling significantly. Currently only branch instructions are monitored to generate a profile of executed basic blocks; profiling data sizes requires monitoring the size of values generated by data processing instructions or gathering snapshots of the size of values stored in the register bank at strategic points within the code.

In addition the size of a value can only be estimated by profiling it dynamically; there is a possibility that values that exceed this estimated size could arise later in the execution of the program. Contingency would have to be built in to the reconfigurable hardware structure to handle these cases. One possibility could be to perform this in a similar fashion to the unexpected branch decision handling: the block terminates, restores state to what it was at the start of the block and continues executing from that point in software.

Without this extra profile information the functional units in the reconfigurable hardware must use the same data size as the microprocessor: in the case of ARM this is 32 bits. If smaller values than the maximum word size will not be detected then it would be pointless to provide a capability for smaller functional units within the reconfigurable hardware structure. All values will be assumed to be 32-bit; a single 32-bit functional unit will be faster and more power efficient than four connected 8-bit functional units when processing these values. In addition a small number of large functional units require less configuration than many small units, reducing configuration size.

5.2.5 Size of compiled block definitions

The compiler generates a *block configuration*, which contains information to configure the functional units and interconnect in each cycle of execution. As hardware blocks are called frequently the hardware must be configured with minimal delay to minimise execution time. To facilitate this, the block configurations need to be stored on-chip, which limits the amount of storage available. Reducing the size of the compiled block configurations will allow more of them to be stored, therefore it would be beneficial for the configurations to take up as little memory as possible. This depends on a number of factors:

Number of functional units Increasing the number of parallel functional units increases the configuration information required for each cycle; however it can reduce the number of cycles required. Therefore the utilisation of the functional units influences the overall amount of configuration data to process a data flow graph.

Flexibility of functional units The number of functional units that can perform a given function should match that operation's frequency in the original object code. By matching the available functions in the hardware to the ones found in the object code the flexibility of the functional units can be reduced to what is most commonly performed. For example, additions are frequently performed and so all functional units should contain an adder to maximise performance. Less frequent operations, such as multiplications, can be restricted to certain functional units. This not only reduces hardware complexity but also requires less configuration information, as the choice of functional unit inherently carries some information on which operations can, and can not, be performed.

Flexibility of interconnect An interconnect that can route data from any location to any other will be very flexible but will require more configuration information than a more restrictive interconnect. Too restrictive an interconnect may adversely affect the ability of data flow graphs to easily fit into the hardware structure, increasing the number of no-op cycles required and decreasing performance.

The structure proposed in figure 5.8 requires 87 bits to configure it for a single cycle[1]. This assumes no data compression techniques are used on the configuration information. These could be used to reduce the size of the configurations: for example if a functional unit is processing a no-op then no further configuration data is required to configure the inputs and outputs to the block, or if no register bank transfers occur during a cycle then the register bank addresses are not required. However, the unit controlling the hardware would have to decompress the information and configure the

[1]ALU operations: 20 bits; Read multiplexers: 20 bits; Register write multiplexers: 7 bits; Register bank write select multiplexers: 2 bits; Register write enables: 8 bits; Register bank write address and enables: 10 bits; Register bank read addresses: 12 bits; Immediate pool address: 8 bits

hardware appropriately, increasing the complexity and power consumption of this control unit.

5.2.6 Implementation of reconfigurable hardware structure

Reconfigurable hardware structures consist of a number of interconnected functional units. The choice of interconnect structure and type of functional units should reflect the structure of the algorithm to be processed and the operations performed. In this system the blocks to be executed consist of data flow graphs created from software instructions. These data flow graphs contain a linear flow of control with no internal loops or branches. The operations performed are restricted to the operations in the original instruction set, such as additions, shifts and logical operations on word-sized data.

An FPGA is therefore unsuitable for the implementation of the reconfigurable architecture for a number of reasons. Firstly, the functional units in FPGAs are typically small look-up tables capable of performing any function on a 4- or 5-bit input, but this level of flexibility is not required in this system due to the limited set of operations that need to be performed and the fixed data width. Large functional units will be faster and more energy efficient than connecting multiple small functional units. Secondly, FPGAs contain a very flexible interconnect that can route data in any direction. As the data flow graphs to be processed are linear the interconnect only needs to be able to route data from the result of one stage to the input of the next. Data does not need to be passed back to previous stages. This allows a simpler, more efficient interconnect than found in FPGAs to be used.

An alternative implementation is to use a reconfigurable pipeline. These consist of a number of stages containing parallel functional units. An interconnect forwards data from one stage to the next. Each pipeline stage is configured with a different stage of the algorithm. This architecture is most efficient if the pipeline is kept full and is therefore suited to streaming algorithms such as audio and video compression, encryption and DSP applications.

The data flow graphs processed in this system only execute a single time each time they are invoked: they do not contain loop structures. A reconfigurable pipeline set up to execute such a block will be heavily under-utilised: only one stage in the pipeline will be in use at any point during the execution of the block. This is highly inefficient in terms of hardware utilisation as most of it will be idle the majority of the time. The ADRES architecture [MVV+03] presents a possible solution to this by tightly coupling reconfigurable hardware with a VLIW processor allowing substantial resource sharing.

Sequenced execution engine

An alternative to the above is to take a single stage of the pipeline and reconfigure it for each stage of execution. The results from one stage of execution are stored in local registers, the hardware is configured for the next stage of execution and the results are fed back in, reusing the same functional units for each stage. The hardware is controlled by a sequencer which is used to configure the hardware structure for each stage of execution.

This structure has a number of advantages over FPGAs and reconfigurable pipelines for this application. The structure more closely matches the structure of the incoming data flow graphs derived from sequential instructions. Larger, faster, more power efficient functional units such as adders and multipliers can be used instead of the relatively small Configurable Logic Blocks (CLBs) of FPGAs. This, in turn, reduces configuration data as there are fewer functional units and a simpler interconnect as there are fewer locations to route data between. Reusing the same functional units for each stage greatly reduces the hardware requirement, improving hardware utilisation and reducing static power consumption.

This architecture bears some similarities to a VLIW architecture; however, there are a number of differences. VLIW architectures require large multi-ported register banks to support multiple parallel functional units, and complex forwarding mechanisms are required to support high performance. This can be partially resolved by partitioning the register file into multiple register banks [CDN94]. In the system proposed in this thesis data values are explicitly forwarded between functional units by a reconfigurable interconnect and held in simple buffers between operations. In addition the configuration data requires little decoding, which reduces decode overhead compared to decoding a stream of instructions.

5.3 Structure description

This section describes a proposed hardware structure for efficient execution of the data flow graphs generated by the hot spot detection and optimisation processes. The structure proposed is based on the design aspects discussed in the first half of this chapter.

Figure 5.8 shows the basic structure of the proposed execution engine. A sequenced execution engine is used, as the flexibility of an FPGA is not required and would be inefficient, and a reconfigurable pipeline structure would be heavily under-utilised. Four functional units are arranged in parallel within an interconnection structure. Each functional unit has two registers in the same column; the result from the functional unit can be written into one, both or neither of these registers. Having two registers per column allows results to be written into one whilst the other preserves a value from a previous calculation. Each input port of the functional unit can read a value from either

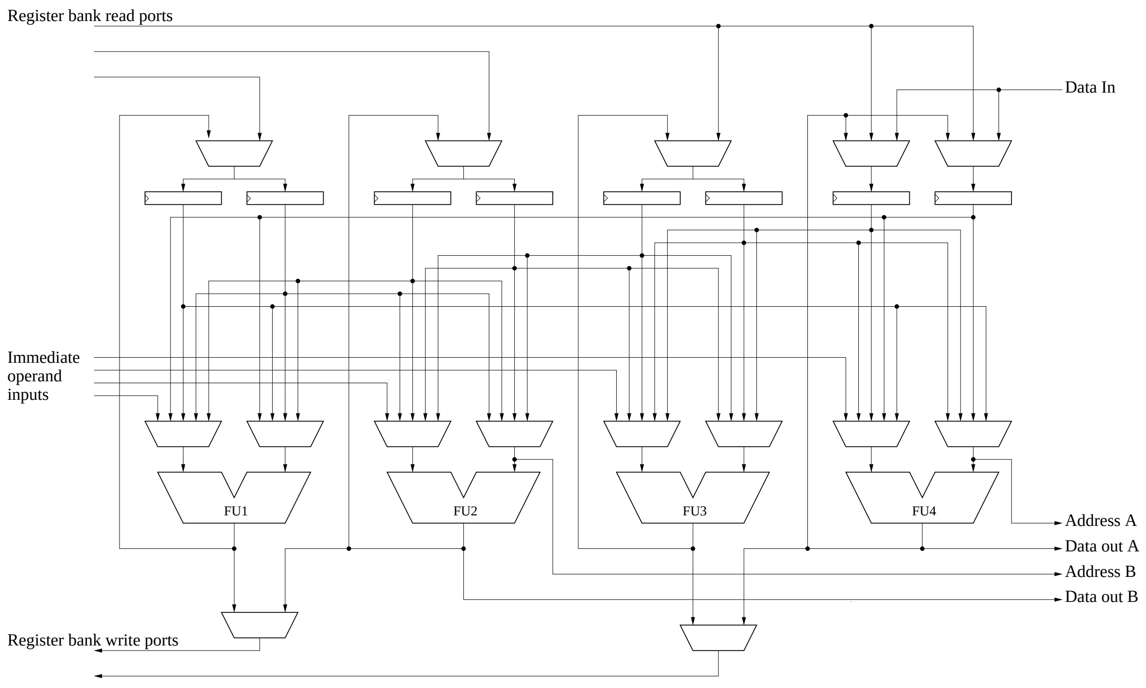


Figure 5.8: Hardware structure internal interconnect

of the two registers in the same column or the adjacent register from either of the neighbouring columns.

A sequencer controls the structure. This sequencer reads the block definition from the Hardware Configuration Table and configures the hardware for each execute cycle. The block configuration data specifies how the interconnect should be configured and which functions should be performed by each functional unit. This operates in a similar method to microcode.

5.3.1 Memory interface

Only a single load may be performed in each execution cycle; this reduces the need for a high-bandwidth interface to memory. Loads can only be performed by one of the functional units, FU4. Loads may take a number of cycles, due to cache and memory latency. Loaded values enter a queue and are transferred into the result register of FU4 under the control of the sequencer. This allows loads to be interleaved. Load requests are scheduled as early as possible

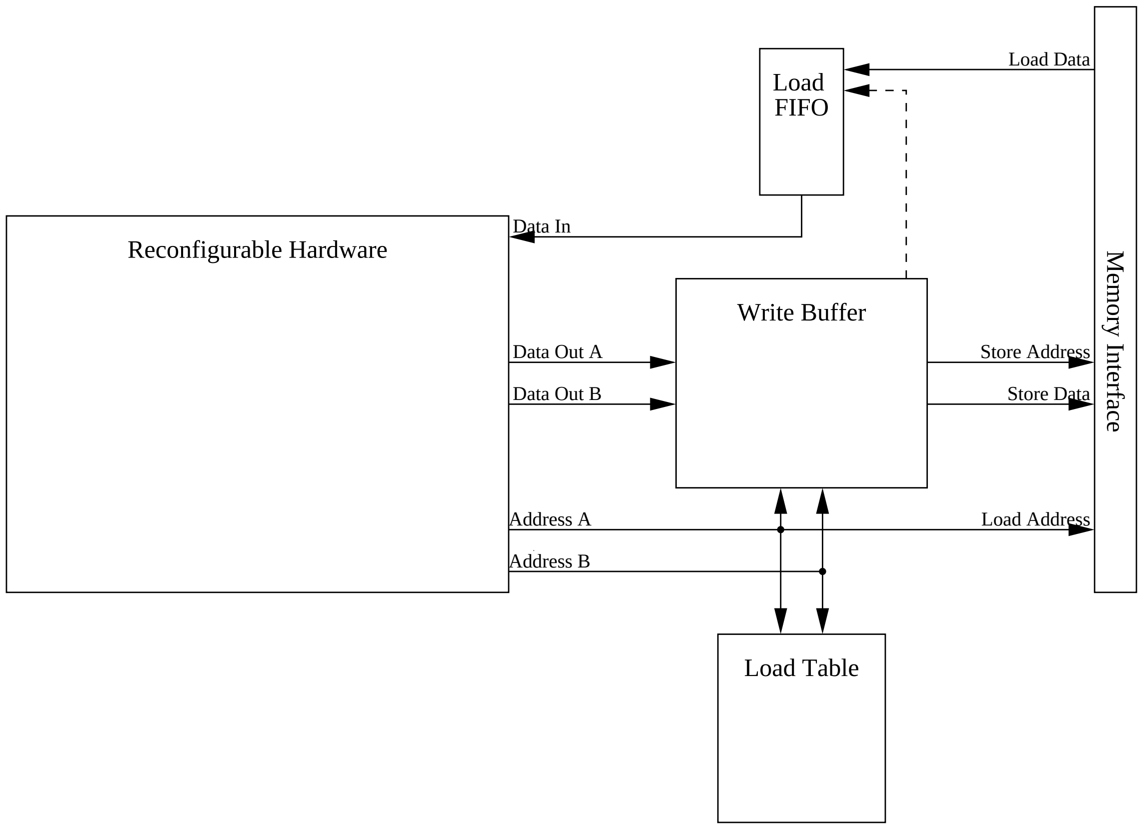


Figure 5.9: Memory interface block diagram

within the block and operations that use these loaded values are scheduled as late as possible, to allow for cache latency. If a load latency is longer than expected (for example on a cache miss) then the hardware structure can be stalled until the load completes. Figure 5.9 shows a block diagram for the memory interface.

Write buffer

Stored values are transferred to a *write buffer* instead of being stored directly to memory. This allows memory state to be kept unaltered until the block completes, allowing state to be easily restored if the block terminates unexpectedly. Upon successful block completion the values in the store buffer are written back to memory. The write buffer also allows multiple store operations in the reconfigurable hardware to be performed per cycle without the need for a multi-ported memory interface. The write buffer can accept two stored values per cycle from the reconfigurable hardware; these are written back to memory one at a time during spare memory cycles after the block completes. This allows improved performance (see figure 5.6) and increases the flexibility of the structure to ease scheduling. The functional units that can perform stores are restricted (to FU2 and FU4, see figure 5.8) to reduce the complexity of the interconnect within the hardware structure. The write buffer is described in more detail on page 123

Detecting memory aliasing and preventing hazards

The use of a write buffer means that multiple copies of a memory location may exist simultaneously. Load operations must check the write buffer for values due to be written to the location being loaded from: if the address being loaded from is present in the write buffer the load should use the value in the write buffer instead of the value in memory as it is more recent. This applies to both loads from the reconfigurable hardware and from the microprocessor, as the write buffer is not flushed back to memory immediately on block completion.

In addition the scheduling of operations during hardware execution means that loads and stores may be executed out of their original order. Instructions that can be determined to alias by the compiler will be scheduled in order. Otherwise, the compiler will assume that there is no dependency between the reordered transfers, so a load may become speculative. The hardware must check for aliasing at run-time to determine whether this speculation is correct. This is achieved by having a *sequence number* for each memory operation in the block; this is assigned by the compiler and corresponds to the order in which the operations appeared in the original code. Three possible types of memory hazard can occur:

Write-after-write The write buffer contains the sequence number for each store operation. Multiple values at the same address may be kept in the write buffer with different sequence numbers: only the last numbered value is written back to memory after the block completes. This ensures only the most recent value at each address is written back to memory, preventing write-after-write hazards caused by out of order store operations.

Write-after-read If a load-store sequence occurs and the store is scheduled before the load, then the load may read the value from the later store operation. A load should therefore ignore any values in the write buffer with a sequence number higher than its own. A load which matches an address in the write buffer will load the store with the latest sequence number before its own sequence number. This ensures the load will not read values from the write buffer from later sequenced stores that have executed before the load.

Read-after-write These occur when a load is moved before a store to the same address. As the value has not yet been stored, a previous value will be loaded. To detect this a *load table* (see page 124) is kept of the address and latest sequence number of load operations executed within the block so far. During a store operation the load table is checked for matching addresses. If a store to a loaded address occurs with an earlier sequence number then a read-after-write hazard has occurred. This occurs infrequently, so at this point the block is broken out of in the same way as if an unexpected branch decision had occurred. An example is shown in figure 5.10.

Alternatively, the hardware could check whether a *silent store* was performed. Silent stores overwrite a memory location with the value already present. These occur frequently [LL00] and would allow the block to continue executing.

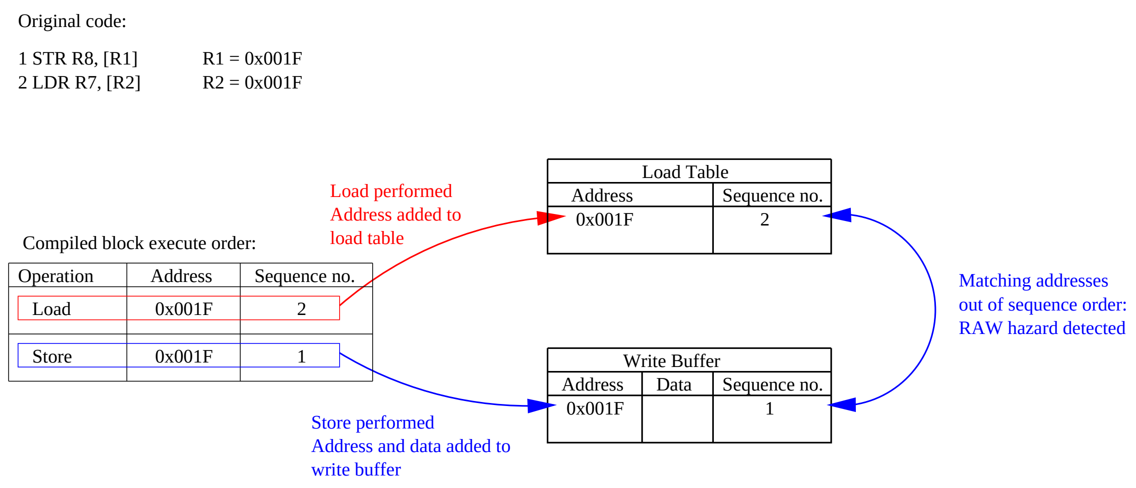
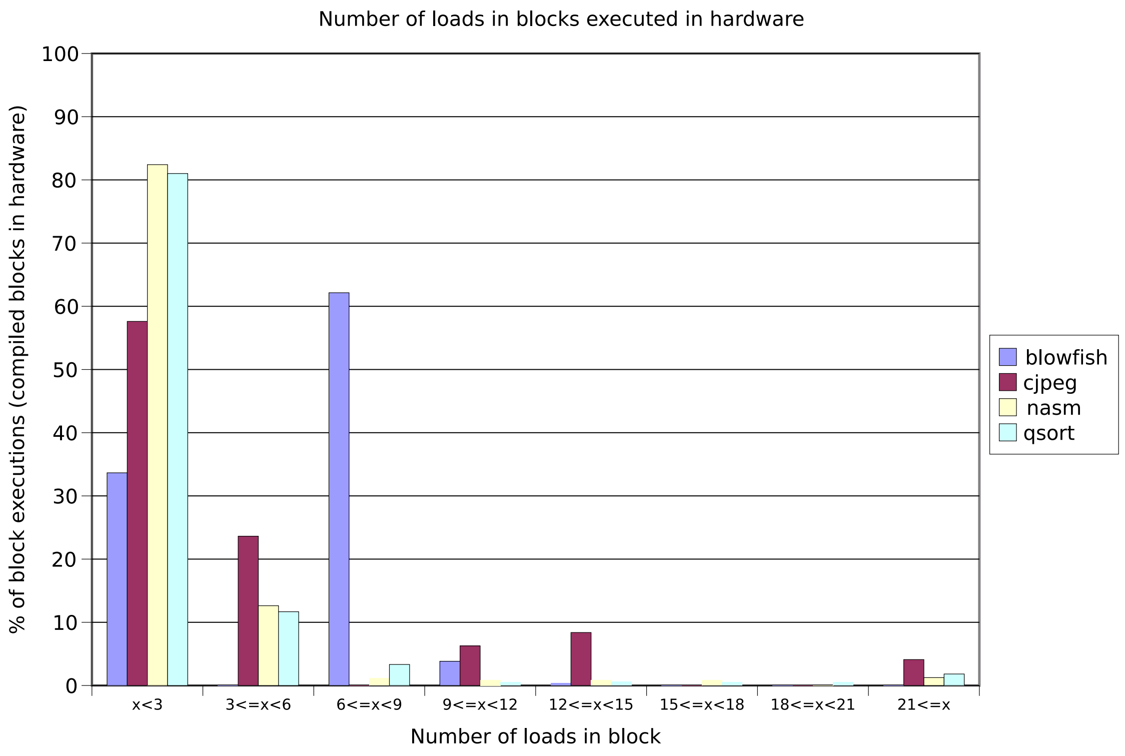


Figure 5.10: Read-after-write memory alias detection

Handling memory hazards from a compiler perspective was discussed in section 4.2.3.

Size of write buffer and load table

The number of loads and stores in each block is known at compile-time; any that contain more loads or stores than available locations in the load table or write buffer can be divided into multiple blocks by the compiler. These would then be treated individually by the hardware, with results being written back to registers and memory at the end of each section of the block. This incurs a performance penalty however: operations will be constrained to one section of the block, reducing parallelism. In addition power consumption will be increased due to the need to write results back to registers and memory between blocks. Therefore, it is advantageous to have a load table and write buffer large enough to allow the majority of detected hot spots to be compiled into a single hardware block. Figure 5.11 shows the distribution of the number of loads and stores in executed hardware blocks. The number of loads in each block is lower than nine in over 90% of all benchmarks except *cjpeg* where about 15% of blocks contain between nine and fifteen loads. The number of stores per block is generally lower than the number of loads, 85% of blocks in



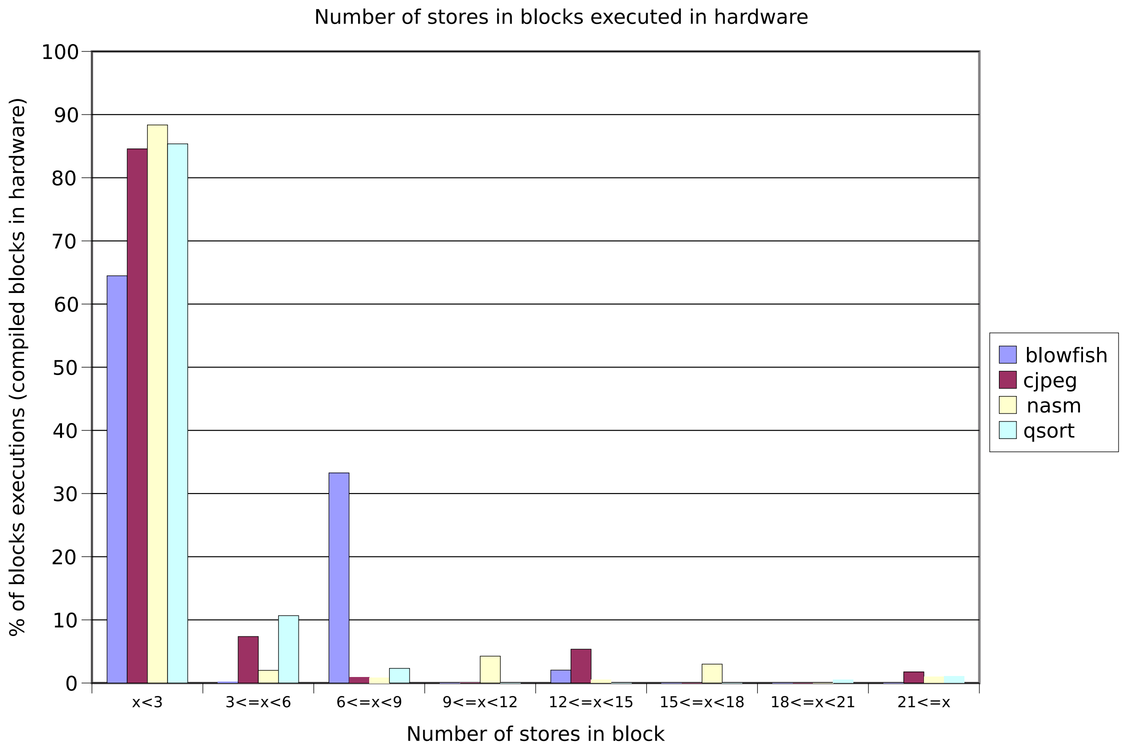


Figure 5.11: Number of loads and stores in executed hardware blocks

all benchmarks except *blowfish* have fewer than three stores per block. These results suggest that the load table and write buffer can be kept small, for example sixteen locations in each, while still allowing most hot spots to be compiled as a single block.

Upon commencing execution of a hardware block the write buffer may not be empty: stores from a previous hardware block may not yet have been flushed to memory. This could be handled by stalling the hardware structure until the contents of the write buffer can be flushed to memory but this would incur a performance penalty. An alternative method is for the block to begin executing and write values to the write buffer as the values from the previous block are being stored, meaning stores from multiple blocks would be present in the write buffer simultaneously. The write buffer hardware must keep track of which stores are in the current block to allow the correct stores to be discarded if the block is terminated, to allow the write buffer to be checked for hazards within the currently executing block and to ensure that results are written back to memory in the correct order.

This is achieved by adding a *completed block* flag to each location in the write buffer: this is set for stores from blocks that have completed. Upon block completion any values from previous blocks to be stored to addresses overwritten by the current block are removed from the buffer and all the completed block flags are set. This marks all values in the write buffer as from a completed block, allowing them to be written to memory. If at any point during block execution there are no free locations in the write buffer then the currently executing block is stalled to allow values from completed previous blocks to be flushed to memory. Note that while the buffer is not empty subsequent memory accesses from code executing in the microprocessor must also check the write buffer to ensure consistency: loads must use not-yet-committed values in the write buffer and stores must invalidate matching addresses.

Store Address	Valid	Completed	Sequence Number	Size	Stored Data

Figure 5.12: Write buffer

Write buffer

The write buffer (figure 5.12) operates as a fully associative table indexed by the address of stores that have occurred. Each entry contains the size of the store, the data stored, the sequence number of the store, a flag to mark whether the store is from a completed block and a valid flag. Stores are buffered until the block completes to prevent memory hazards as described on page 118. The size information is required to allow partial stores to be correctly stored in the write buffer and merged with any matching loads that occur. The operation of the write buffer during events that affect it is described below:

Store: When a store occurs an entry with the sequence number and the data to be stored is added to the write buffer tagged by the address to be written. The entry is marked as in the current block.

Load: During a load address tags in the write buffer are checked for matching addresses. Sequence numbers of any matches are checked against the sequence number of the load. If an entry exists with an earlier sequence number than the load's sequence number then the data value stored is returned as the loaded value (If there are multiple matches the highest sequence number not greater than the load's sequence number is used). If there are no matches with earlier sequence numbers then the value is loaded from memory.

Block completion: All entries are marked as from completed blocks when a block completes. Any writes from previous blocks to addresses written to by the completing block are marked invalid to reduce unnecessary memory writes when flushing values.

Block breakout: If a block is broken out of before completing then all entries not already marked as from completed blocks are marked invalid.

Writeback: Values marked as from completed blocks are cleared from the buffer and written back to memory during cycles when the memory bus is free. The buffer is checked for matching address tags. If duplicate writes from completed blocks to the same address are found then only the value with the latest sequence number is written back to memory; the others are discarded by marking them invalid.

If the table becomes full execution is stalled to allow values from completed blocks to be written back. As the compiler knows the size of the write buffer it can limit the size of compiled blocks if the number of stores reaches this size, ensuring that the write buffer can always at least hold all of the stores in any compiled block.

Load table

The load table maintains a list of addresses that have been loaded from in the currently executing block. The latest sequence number for each address is stored to allow read-after-write hazards caused by out of order execution of loads and stores to be detected. The load table is structured similar to a small fully associative cache: the address is used as the index into the table. During relevant events the load table operates as follows:

Load: When a load occurs the address of the load and its sequence number is added to the load table. If the address already exists in the table the sequence number is updated if the currently executing load has a later sequence number than already stored in the table.

Store: During a store operation the load table is checked for a matching address. If a match is found the sequence number is compared with that of the store; if it is later than the store then a read-after-write hazard has been detected and the executing block is broken out of.

Block completion or block breakout: The load information for a block is no longer needed after it exits and so the table is cleared.

As with the write buffer, the load table cannot become full during the execution of a block as the compiler is aware of the size of the write buffer and so can limit the size of a compiled block to prevent it containing more loads than the table can hold.

5.3.2 Interface with architectural registers

Although the number of registers that must be read during a block is small, the operations in the first cycle of a block will require values from the register bank before they can execute. This means multiple parallel transfers from the register bank to the reconfigurable hardware must be possible to allow the block to begin executing as soon as possible. A microprocessor register bank will typically contain multiple read ports to supply software instructions with multiple operands; these can be used to allow multiple transfers to the hardware structure.

Three registers may be transferred in each cycle, requiring three register read ports. The third register read port is shared between FU3 and FU4, meaning that only one of these can read a register in the same cycle.

Transfers from reconfigurable hardware to architectural registers happen more frequently; however, they are typically more evenly distributed throughout the block meaning that fewer parallel transfers are required. Two register write ports are deemed sufficient. Register renaming is used to preserve register state at the start of the block.

5.3.3 Functional units

As discussed in section 5.2.4 the size of the functional units should match that of the microprocessor instructions being compiled into hardware. In this case an ARM microprocessor architecture is used, which has a 32-bit operand width. In addition to matching the width of the data to be processed, the operations that a functional unit can perform should match those that will be contained in the data flow graphs generated from the object code of hot spots. The set of possible data processing operations is small: it consists of additions and subtractions (including compares), bitwise logical operations, shifts and multiplications. This limited set of operations means that the flexibility of FPGA-like look-up tables is not required, allowing faster and more power efficient fixed-function ALUs to be used.

The number of functional units that can perform certain operations should match the frequency of those operations in the generated data flow graphs. Although four functional units are present within the hardware structure they need not all be able to perform every operation. Figure 5.13 shows the relative proportion of different types of data operations in compiled blocks. The majority of operations are arithmetic: these include additions, subtractions, compares, and address additions or subtractions. These operations are very common and can all be performed with an adder; therefore all functional units should contain an adder to ensure performance is not limited by a lack of adders. In contrast, very few multiplications are performed in these benchmarks. By only having one functional unit with a multiplier the size and power consumption of the hardware structure can be reduced (as multipliers are relatively large) without significantly affecting performance. However, if this system was targeted towards applications that frequently use

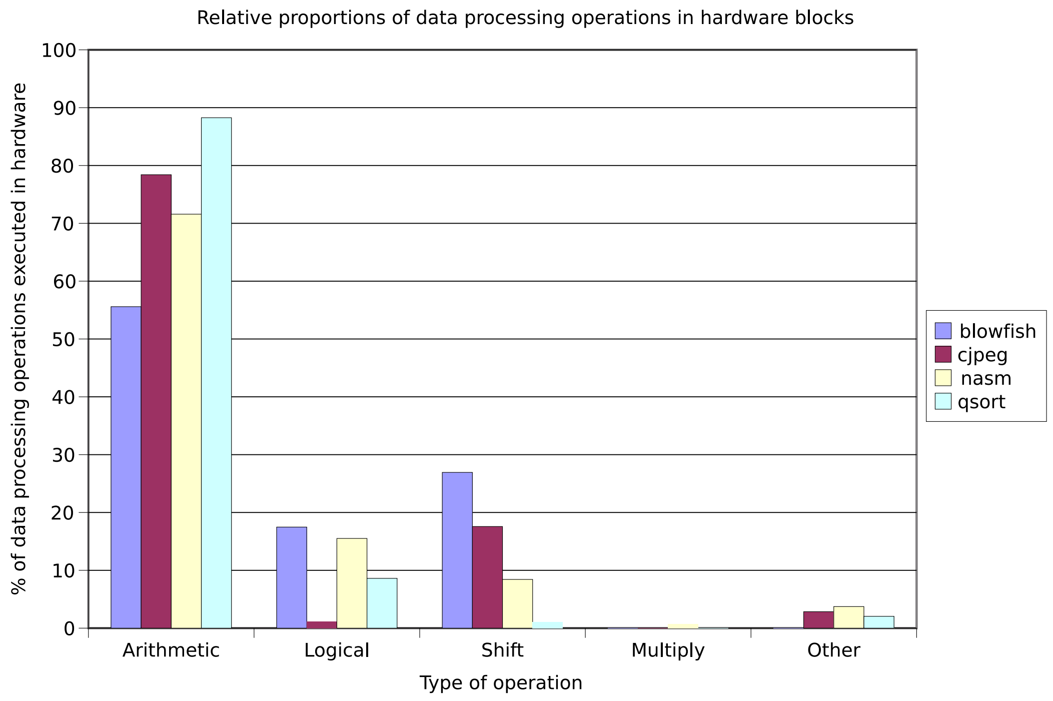


Figure 5.13: Proportion of data processing operation types in executed hardware blocks

multiplications, such as DSP algorithms, then having a single multiplier may limit performance.

5.3.4 Immediate operands

Most instruction sets contain mechanisms for embedding data in the instruction stream. These are called *immediates* and provide convenient access to constants within a program.

Immediates do, however, create difficulties when the instructions are to be compiled into a hardware structure. The instruction stream, which contains the immediate values, is not fetched when executing in hardware and so an alternative method must be used to supply the functional units within the hardware structure with these values. Immediate values could be stored within the compiled definition of the block and 'immediate' registers within the hardware structure configured directly for each execution cycle by the

blowfish		cjpeg		nasm		qsort	
Value	%	Value	%	Value	%	Value	%
1	13.7	1	21.9	1	26.2	4	30.6
2	11.7	2	10.1	0	19.2	1	24.9
8	11.3	0	8.7	4	8.2	0	22.0
FF	10.2	4	7.6	FF	6.1	8	3.3
10	8.7	3	3.4	3	4.7	3	2.2
4	6.2	6C	2.7	2	4.5	FF	2.2
0	4.0	30	2.5	1F	2.8	10	0.9
48	2.9	10	2.3	8	2.6	2	0.7

Table 5.1: Most frequent immediate values, sorted by percentage of total immediates within hardware blocks

sequencer. This would require a large number of configuration bits, as a 32-bit value would be added to each functional unit's configuration data.

Immediate values are frequently re-used multiple times during a program's execution. Table 5.1 shows the most frequent immediates used within compiled blocks in the benchmarks. Some values are frequently re-used, such as 1, 4 and 0, as they are commonly used in increments, address offsets, and compares. Storing immediate values in an *immediate pool* allows values to be specified once and then re-used multiple times. Values are added to the pool as the blocks containing them are compiled and loaded into the HCT. Values can be overwritten when they are not contained within any currently loaded block.

The immediate pool should be large enough to contain all the immediates currently contained within the set of loaded blocks. Table 5.2 shows the maximum number of different immediate values contained in simultaneously loaded blocks. *Cjpeg* requires 113 different immediate values to be simultaneously available, requiring a large immediate pool. The immediate pool must have multiple read ports to support parallel immediate operand access; this, combined with the large number of required entries, would require a large amount of hardware, increasing power consumption. In addition, if the

Benchmark	blowfish	cjpeg	nasm	qsort
Max. loaded immediate values	24	113	64	54

Table 5.2: Number of different immediate values in blocks stored in HCT (maximum at one time)

number of required immediate values became larger than the immediate pool some compiled blocks would have to be discarded to free space for immediate values in newly compiled blocks.

An alternative would be to only load the immediates used in the current block into the immediate pool. These would be loaded from the block configuration as each block was entered. This would require a much smaller immediate pool: figure 5.14 shows that the vast majority of blocks contain fewer than 16 different immediate values. The compiler could divide blocks that contained a greater number of immediates into multiple blocks (as is done with blocks containing too many memory operations). However, the number of values that must be loaded into the immediate pool at the start of each block is quite large, with many blocks requiring four or more different immediate values. This increases the amount of data that must be transferred from the HCT during each block execution.

Table 5.1 shows that many immediate values, such as 0 and 1, are used commonly in all benchmarks. By keeping a set of these permanently available in the immediate pool the number that must be loaded from the block configuration data at the start of each block can be reduced. Figure 5.15 shows the number of immediate values required when eight values are kept permanently within the immediate pool. The values used are the same in all benchmarks. The majority of blocks require three or fewer additional immediates to be loaded from the block configuration data; over 80% of blocks in *nasm* and *qsort* require none at all. An immediate pool size of 8 configurable values, in addition to the 8 fixed values, is sufficient in most cases, This hybrid arrangement of predetermined and configurable immediates exploits commonly used immediate values to reduce the amount of configuration data required, and therefore the size and power consumption of the HCT.

5.3.5 Control of block execution

Operations within the block are scheduled statically to execute in parallel using the available functional units. As the operations to be performed in each cycle have been pre-determined by the compiler all the sequencer needs to do each cycle is configure the functional units and interconnect within the hardware structure based on the compiled configuration data. No conditional branches are contained within the block, so operations execute in a linear sequence. Controlling the hardware structure is therefore straightforward: much of the work is done at block compile time.

Some dynamic control of operation execution is required however. Instructions in some architectures may be executed conditionally. Points where conditional branches have been removed by the basic block chaining process must be tested to determine whether the block can continue executing or must be terminated. In addition, the address of the next instruction to be fetched after the hardware block has exited must be loaded into the program counter to allow execution to continue.

Conditional instructions

In the ARM instruction set any instruction may be executed conditionally: each instruction contains a 4-bit condition field that determines which, if any, of the condition flags are used to decide whether the instruction executes. The hardware structure must also be able to perform conditional execution of the operations contained within conditional instructions. This requires being able to select between two possible values for a register depending on the condition: the value written if the conditional operation is executed and

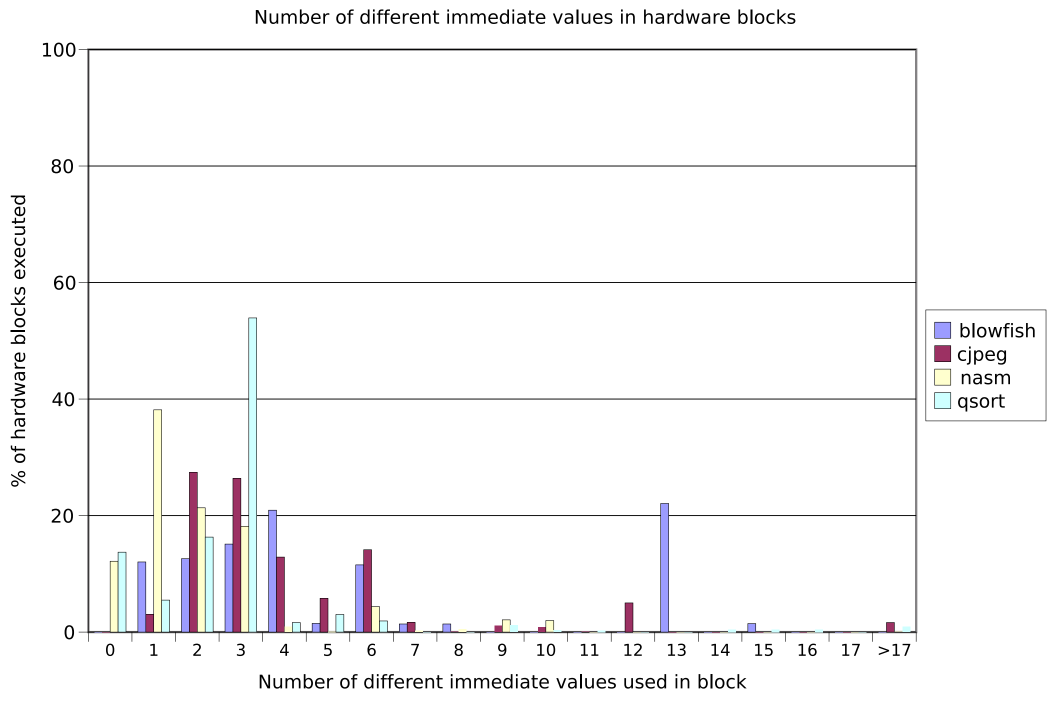


Figure 5.14: Number of different immediate values in each executed hardware block

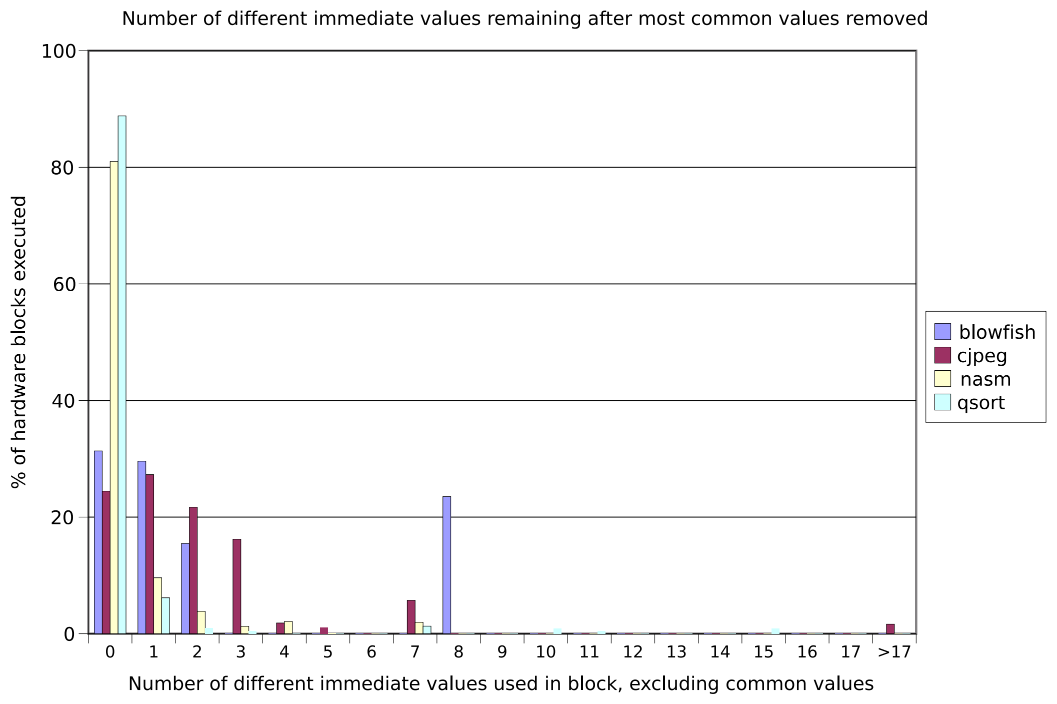


Figure 5.15: Number of different immediate values in each executed hardware block, excluding most common values (0, 1, 2, 3, 4, 8, 10 and FF)

the original value to be used if the operation does not execute. This can be achieved by scheduling operations in such a way that the original architectural register maps to a hardware structure register. The result from the conditional operation can then overwrite this value if the condition is true; if the condition is false then the previous value remains in the register.

Detecting and handling block exits

During the block chaining process branches at the end of basic blocks are removed; these may be conditional branches. It is assumed, based on profile information, that these branches will be taken in a single direction the majority of the time but the condition tests must still be performed to ensure the program executes correctly. Operations that test the conditions for these branches signal the control unit if an 'unexpected' result occurs. The control unit then terminates execution of the block, removes any values written to the write buffer and hands control back to the microprocessor at the address of the first instruction in the block. The block is then executed in software, allowing the unexpected conditional branch to be taken.

If the block terminates normally the control unit must determine the address of the next instruction to be fetched and write it into the program counter to allow execution to continue. This address may depend on the result of a conditional branch at the end of the last basic block. The branch destination addresses of a final branch must therefore be stored within the block configuration data. In some cases the next address may be generated by the operations executing within the hardware block: for example if the block terminates with a stack pop operation then the address will be loaded from memory and so need not be stored in the configuration data. If this address matches the start of a compiled block then that block is loaded into the hardware structure and execution continues in hardware. Values are transferred between consecutive hardware blocks via the software register bank to allow the later block to break out and restore control to the microprocessor at its start point if necessary.

5.4 Block mapping algorithm

Section 4.3 described the process of constructing a block from a chain of basic blocks and producing an optimised data flow graph (DFG) of the block. A mapping algorithm is required to map the operations in the DFG into the hardware structure taking into account the restrictions on operation and data memory parallelism, the limited capabilities of individual functional units in the hardware structure and the restrictions in the interconnect between functional units.

The algorithm used to schedule the DFG into the hardware structure first prioritises operations in the DFG based on the number of dependent operations and the length of the path they are in. Operations at the start of the critical path of the block will therefore be given the highest priority. Operations that generate a result used by many other operations in separate paths are also given a high priority. The scheduling algorithm attempts for each execution cycle in the block to schedule the remaining operations with the highest priority into the available functional units, taking into account the restrictions on the number of parallel operations of certain types, such as stores. Load operations are also scheduled as early as possible before the loaded value is required to reduce the effect of a stall caused by a cache miss.

The results in this thesis were generated using this simple mapping algorithm to estimate the number of cycles required to execute a block in the hardware structure. This algorithm, however, does not account for the restricted routing of the interconnect between execution stages (see figure 5.8). To verify the validity of this simplified model for generating results, a selection of the most frequent blocks in the benchmarks were hand-mapped

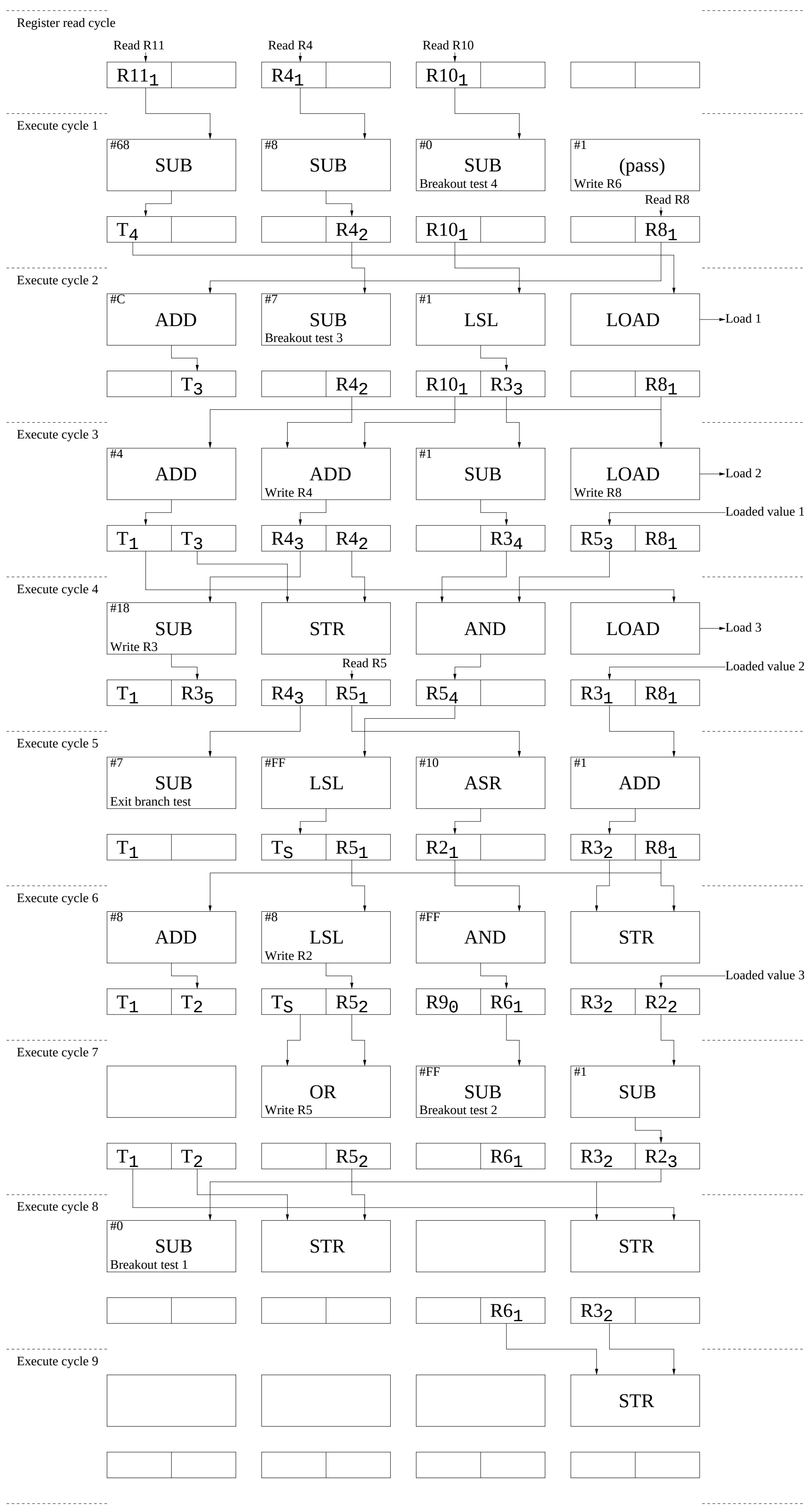


Figure 5.16: Example mapping of block on page 77 into hardware structure

into the hardware structure following the restrictions of the interconnect. The measured cycle time of these hand-mapped blocks was compared to the estimated cycle times of the blocks mapped using the simplified algorithm. The simplified model matched the hand-mapping in most cases due to the length of the critical path through the block limiting the available parallelism to less than four and so reducing the impact of the limited interconnect. Some blocks with a large available parallelism or short critical path, however, can not complete in the same number of cycles as the simple mapping estimate due to the restricted interconnect.

Figure 5.16 shows an example of this; the figure shows a block hand-mapped into the proposed structure, following the restricted interconnect. The block used is the same as the example in chapter 4 and shown on page 77. The block takes nine execute cycles to complete, with an additional cycle required at the start to read register operands. The last execute cycle contains a single store operation. The constraint that stores can only be performed by FU2 and FU4 restricts the scheduling of operations, preventing this store from executing in an earlier execute cycle despite the required operands being present.

5.5 Summary

This chapter outlines the requirements of a hardware structure suitable for executing the data flow graphs generated by the processes in chapters 3 and 4. Three is determined to be the most efficient number of functional units from a performance and utilisation perspective; however four functional units will ease the scheduling of operations and allow slightly higher performance without severely impacting utilisation. A single load per cycle allows adequate performance while allowing for a simple memory interface.

A suitable structure is proposed: this is a sequenced execution engine containing four parallel functional units and eight accumulator registers.

An immediate pool is used to supply immediate operands to the hardware structure. Operations within blocks may be executed speculatively due to block chaining, requiring mechanisms to preserve architectural and memory state while the block executes. A write buffer holds stores until the block completes, and register renaming is used to preserve register values. Mechanisms for detecting memory aliasing are also described.

Chapter 6

System Analysis

The previous three chapters described the processes of hot spot detection, compilation and execution in hardware separately. This chapter combines these three elements to analyse the system as a whole. The first half of this chapter discusses performance improvements gained by performing dynamic compilation to hardware, while the second half discusses possible effects on power consumption compared to existing architectures.

6.1 Evaluation method

To generate these results the ARM simulator described in section 1.2 was modified with a software model of the the hot spot detector described in chapter 3. A 64 entry model of the BPT was modelled, with other parameters as detailed in section 3.5. The model of the HCT used also had 64 entries with other parameters such as breakout limits as detailed in section 3.6. This model extends a software model of an ARM microprocessor. The compiler performs all the optimisations described in chapter 4 with the exception of removing stack push and pop operations contained within blocks. The compiler generates a DFG of each block that can be used to estimate the increase in performance of the block. This DFG is constrained by the 4-way parallelism, single load per cycle, two stores per cycle and single cycle load delay described in chapter 5. The scheduling of operations into the hardware structure to determine parallelism and therefore performance is as described in section 5.4.

6.2 Performance

The main aim of compiling hot spots into reconfigurable hardware was to improve performance. This is primarily achieved by introducing parallelism in blocks. The overall performance increase is determined by the proportion of instructions that execute in hardware, the increase in parallelism within compiled blocks and the compilation time.

6.2.1 Amount of code executed in hardware

The key determinant of how effective this system is at improving performance is the proportion of instructions executed that are detected as hot spots and compiled into hardware. Parallelisation and other optimisations are only performed on detected hot spots, therefore a larger proportion of blocks executed in hardware will increase the effect of performance gains.

Code sections can only be executed in hardware once they have been detected as hot spots, compiled, and loaded into the hardware structure. Figure 6.1 shows the proportions of instructions executed during a dynamic trace which are executed by the reconfigurable hardware structure. These are equivalent numbers of software instructions: the original instructions are no longer fetched and executed as they have been translated into a different form. The remainder are instructions not determined to be within hot spots and so are executed in a standard fashion within the microprocessor pipeline.

These results show that for *blowfish*, *cjpeg* and *qsort* the majority of instructions – 92% to 96% – are executed in the reconfigurable hardware.

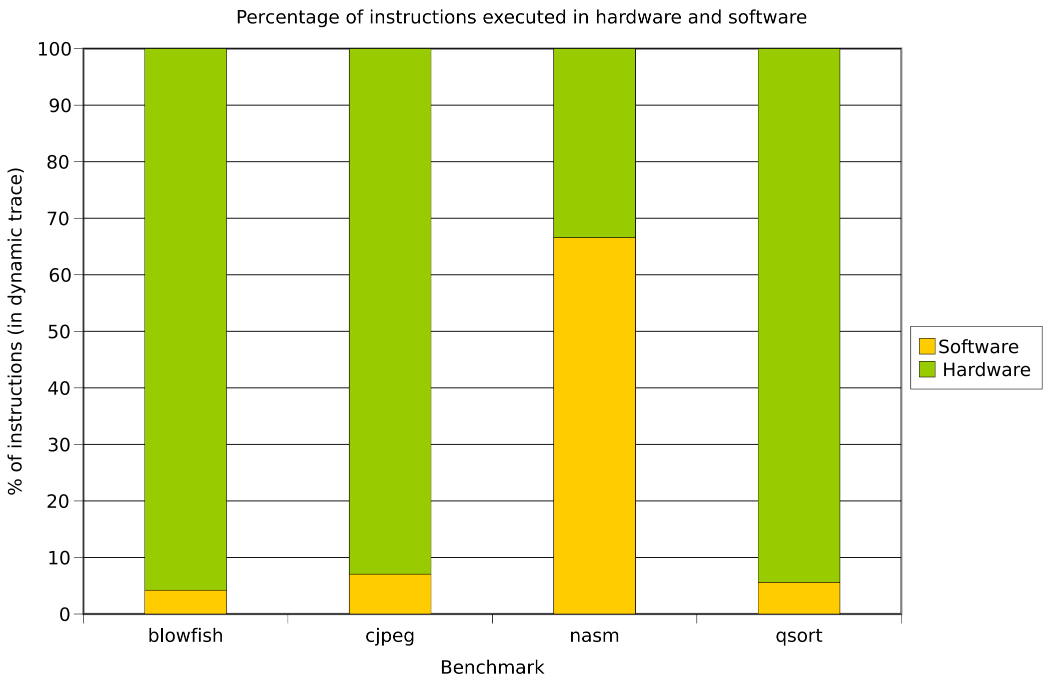


Figure 6.1: Percentage of instructions executed in hardware and software during benchmark execution

As expected, a much smaller proportion of instructions in *nasm* execute in hardware, due to this benchmark containing no significant hot spots.

6.2.2 Effect of block breakouts

If a block must be terminated due to an unexpected branch decision or an unresolvable memory hazard then results generated so far within the block are discarded and execution restarts from the start of the block in software. This causes some extra operations to be performed, of which the results are not used. This has an impact on overall performance. Figure 6.2 shows the number of instructions executed in blocks that are broken out of. In a worst case scenario, where all operations in a block are executed before a breakout is detected, all of these instructions will effectively be executed twice: once in hardware and once in software. However, as the number of instructions

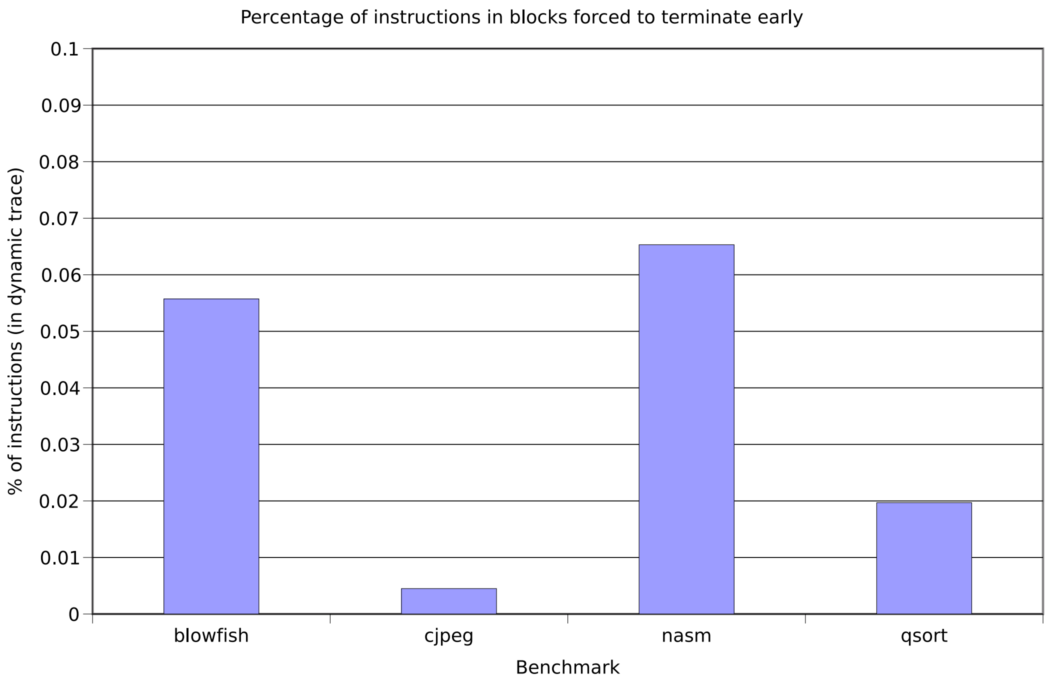


Figure 6.2: Percentage of instructions re-executed due to hardware blocks terminating early (upper bound, assuming all instructions in a block are executed twice if it terminates early)

executed in blocks that are terminated early is extremely small the effect of breakouts on performance can be considered negligible.

6.2.3 Performance of blocks executed in hardware

The performance of blocks executing in hardware is generally greater than executing them in software. This is due both to an increase in parallelism and the removal of operations, such as intermediate branches, moves, stack pops, operations duplicated in chained basic blocks and operations only required by not-taken branches. Figure 6.3 shows the gain in performance of blocks executed in hardware, compared to executing them in software. These results are only for blocks detected as hot spots and compiled into hardware, and show a weighted mean based on the number of times each block executes. Again, the results were generated by running the benchmarks in a software

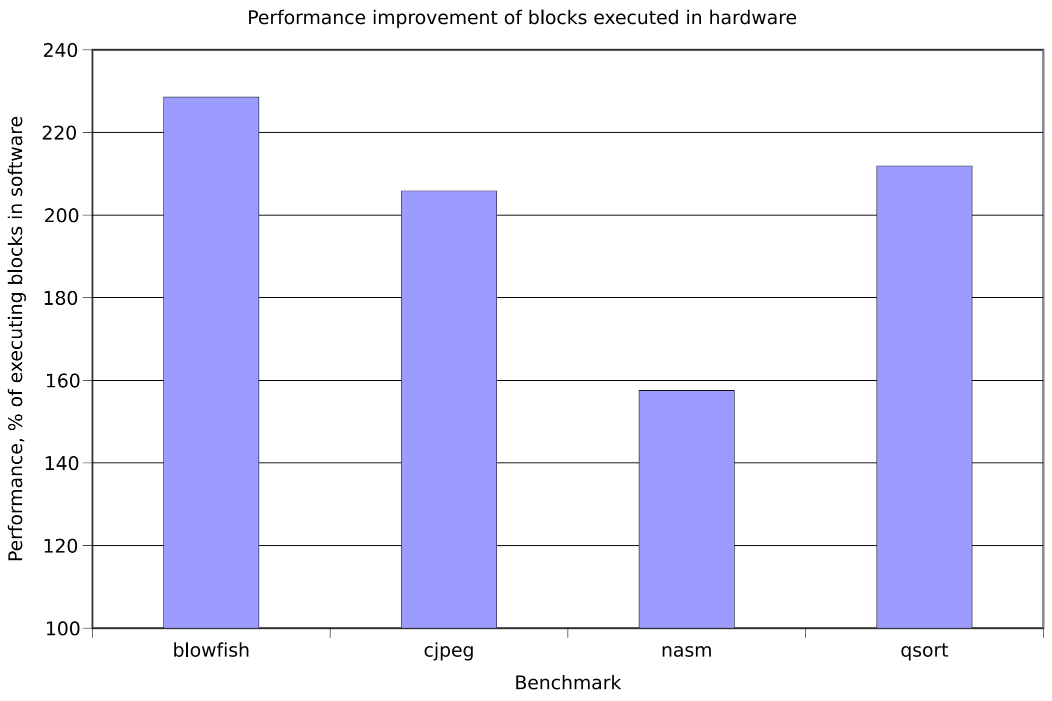


Figure 6.3: Performance improvement of blocks compiled into hardware, weighted mean based on dynamic frequency of execution

model of the system. Blocks detected as hot spots were mapped into a model of the hardware structure as described in section 5.4. This was then used to work out the performance increase of these blocks . The performance increase of each block was then multiplied by the number of times it executed, to give an overall performance increase of all compiled hardware blocks during the execution of the benchmark.

An improvement of over double the software performance is achieved in these blocks in *blowfish*, *cjpeg* and *qsort*. Again *nasm* performs relatively poorly, with compiled blocks performing at 158% of their performance in software. Less predictable execution in this benchmark reduces branch bias, therefore reducing the size of block chains that can be constructed. This has an impact on the available parallelism within blocks and reduces the scope over which other optimisations can be performed.

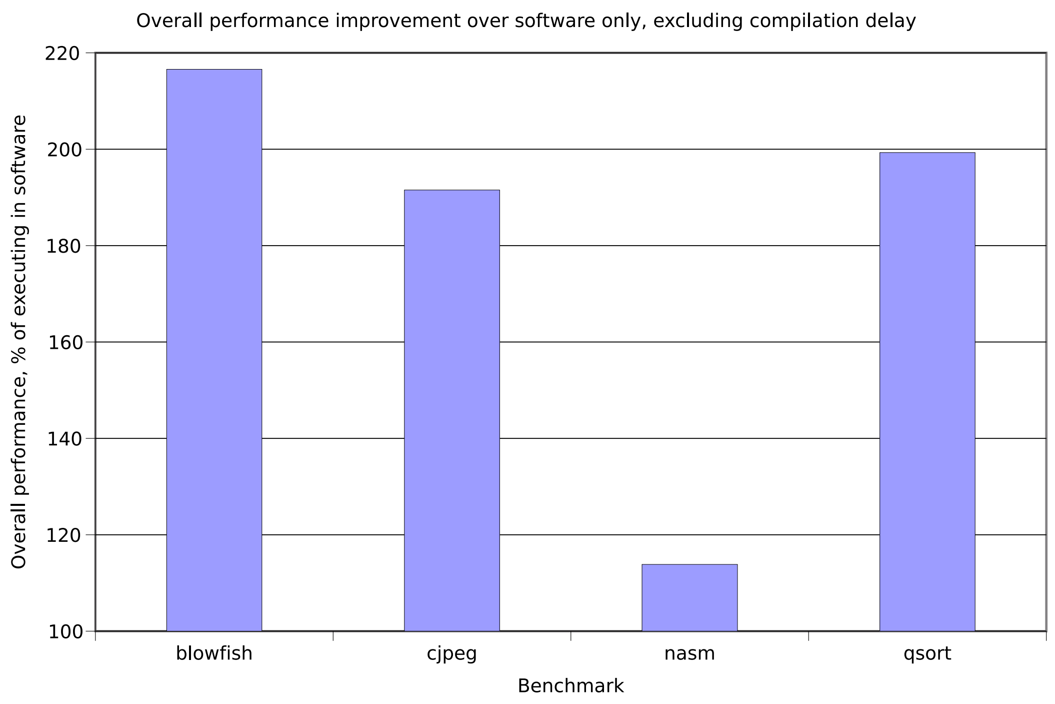


Figure 6.4: Overall performance improvement of benchmark, excluding performance penalties due to compilation process

Figure 6.4 combines the performance improvement results in figure 6.3 with the percentage of program coverage results in figure 6.1. This shows the overall performance improvement due to blocks executing in hardware, over the execution of the entire benchmark. As expected, the impact of the performance improvements due to parallelism and optimisation are reduced, compared to the performance improvement in just the hardware blocks, as not all blocks execute in hardware. The performance improvement over a software system is still significant in *blowfish*, *cjpeg* and *qsort*, as a large proportion of executed instructions are within compiled hardware blocks. *Nasm* has the fewest blocks executing in hardware and shows the smallest performance gain in these blocks. This translates to a small performance increase of about 15%.

The results in figure 6.4 only show performance *improvements* due to the execution of hot spots in hardware. It does not account for any performance overheads caused by the compilation process itself.

6.2.4 Performance impact of run-time compilation

In section 4.4 three possible methods were described for performing the compilation. The first, pausing execution and compiling in software, will cause a delay each time a compilation is performed. The other methods, using the reconfigurable engine or dedicated hardware, will incur less or no delay as execution can continue during compilation but require additional hardware.

Figure 6.5 shows the average number of compilations performed for each million instructions executed. This is very low: the worst case, *nasm*, has only ten compilations for each million instructions executed. Although the overhead of performing compilations cannot be known without an accurate implementation of the compiler software or hardware, these results suggest that the overhead from each compilation could be quite large, in excess of an equivalent 10000 instructions, before it is greater than the performance gained. Using dedicated hardware can inflict almost no performance penalty: other systems that perform run-time translation into VLIW instructions [NH97] and reconfigurable hardware [Lev05] are able to perform similar tasks with very little delay by using dedicated hardware.

6.2.5 Performance analysis

The results presented here show a performance improvement of around double for programs which contain a small number of frequently executed hot spots. Around 95% of instructions in these programs are executed in hardware and compiled blocks execute at slightly over double their performance in software. The performance of *nasm* is only improved by around 10%. This is partly caused by relatively few instructions being executed in hardware, but also due to smaller amounts of available parallelism in compiled blocks.

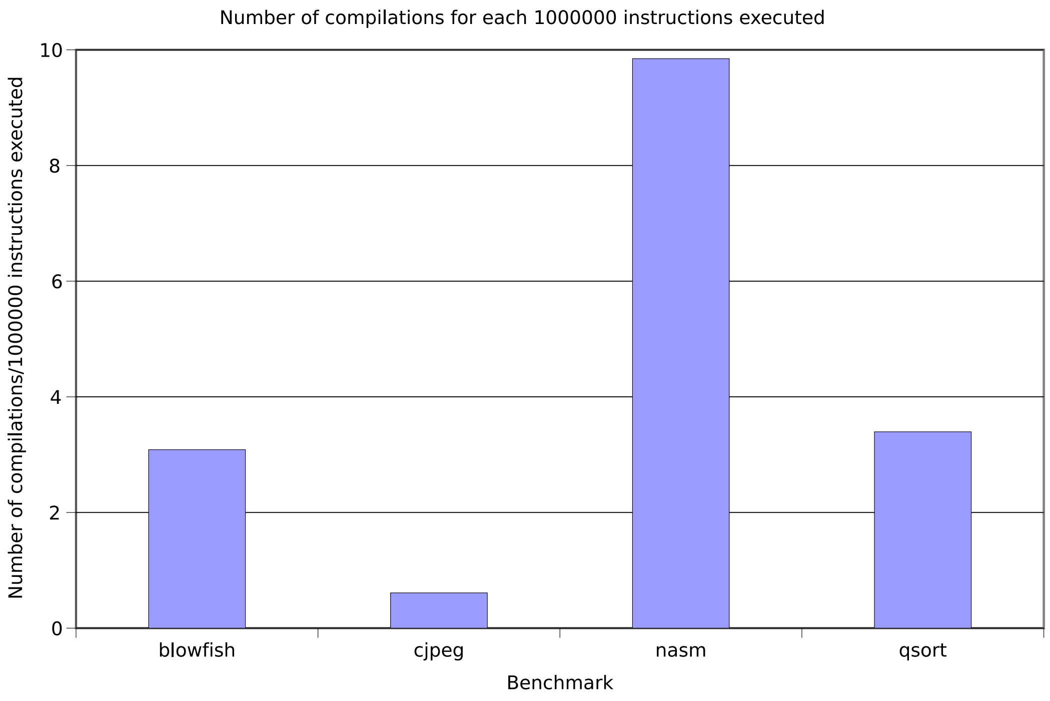


Figure 6.5: Number of compilations per million instructions executed

6.3 Energy consumption

The previous section demonstrates that this system has the potential to improve performance over a scalar microprocessor, primarily by increasing parallelism. Many other architectural techniques, such as superscalar or VLIW, can achieve performance gains by increasing instruction level parallelism. The architecture proposed in this thesis has advantages over VLIW in that it does not require code to be compiled for a specific level of parallelism and is compatible with existing compiled code. Superscalar has these advantages over VLIW; however, superscalar architectures consume more energy than scalar architectures or VLIW due to the requirement to detect dependencies and schedule operations in parallel each time a section of code is executed. This section explores the possibility for the proposed architecture to improve performance without this increased energy consumption.

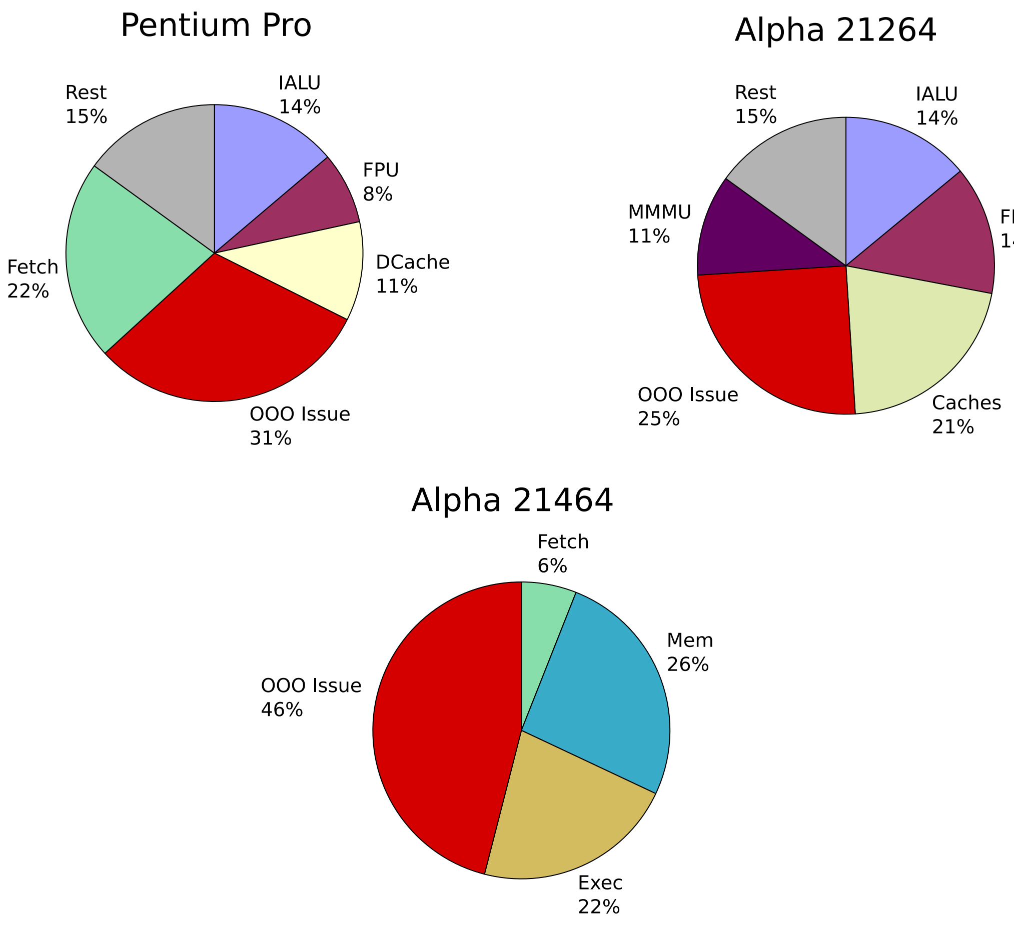


Figure 6.6: Power breakdown of superscalar processors [Val05]

6.3.1 Parallelising instructions

The hardware that performs the dynamic scheduling of operations in superscalar microprocessors accounts for a significant proportion of the total power consumed by the processor [Val05][KGPK01][MKG98]. Figure 6.6 shows the power breakdown of a selection of superscalar architectures; the energy consumed by the Out-Of-Order (OOO) issue hardware which parallelises operations consumes between 25% and 46% of the total power consumed by the processor.

In the architecture proposed in this thesis the parallelising of instructions only occurs for hot spots as these are compiled; once performed the parallelised hot spot can execute many times without the need for this work to be repeated. This greatly reduces the amount of work required to parallelise operations; therefore this architecture may have the potential to be more energy efficient than superscalar architectures. However, power savings may be offset by the additional hardware required to monitor code execution and compile detected hot spots. In addition, some hardware to support out-of-order execution is still necessary, such as the write buffer and load table required to detect memory aliasing due to speculatively re-ordered load and store operations.

6.3.2 Reduced register bank access

The register bank in a microprocessor is one of the most power consuming components: for example it consumes around 25% in the AMULET3 ARM microprocessor core [Eft02]. The need for multiple read and write ports in the register bank increase its size and power consumption over individual registers. The register bank is frequently accessed, typically multiple times in each instruction to read and write operands. Many values stored in the register bank are short lived temporary values [FS92]. This property can be exploited: one method is to store new values for a short time in a buffer, preventing short lived values from being written to the register bank. This has been shown to give a power saving of 30% in one superscalar processor [HM00].

In the system described in this thesis, values in the hardware structure are not forwarded between operations via a register bank. Results from one operation are explicitly forwarded to the input of other operations by the interconnect in the hardware. Any values that are local to the compiled block are not written to the register file: only values that are still valid at the end of a block are written back when the block terminates. This exploits the temporal locality of values to reduce register bank accesses significantly.

Figures 6.7 and 6.8 show the breakdown of remaining register bank reads and writes when executing hot spots in hardware. “Software” accesses are

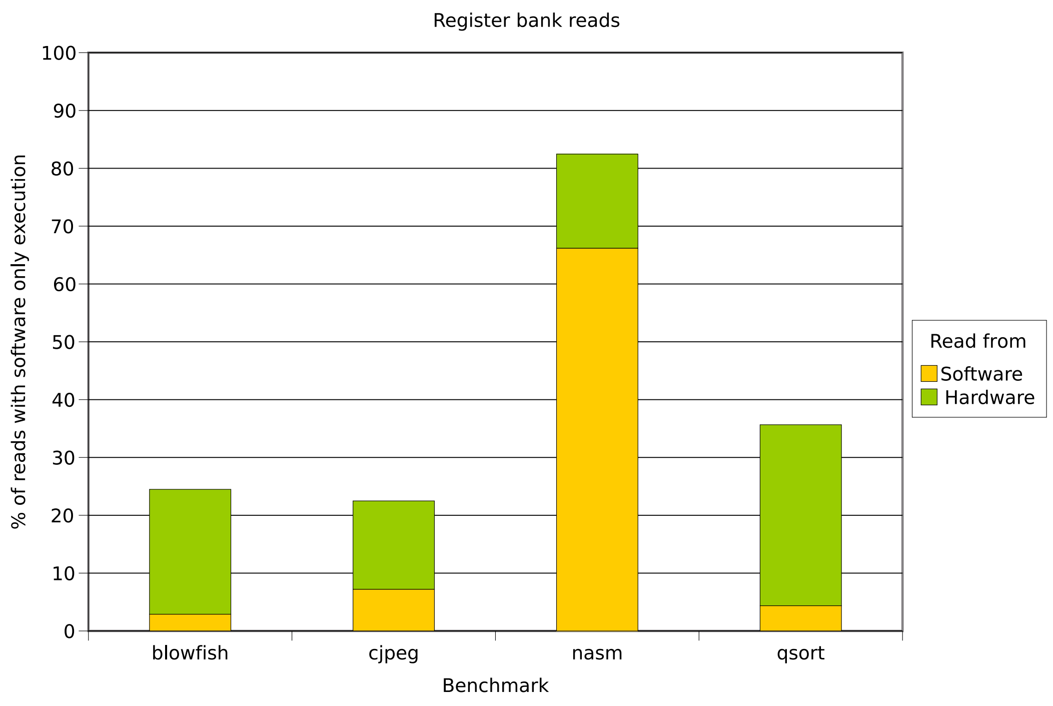


Figure 6.7: Remaining register bank reads

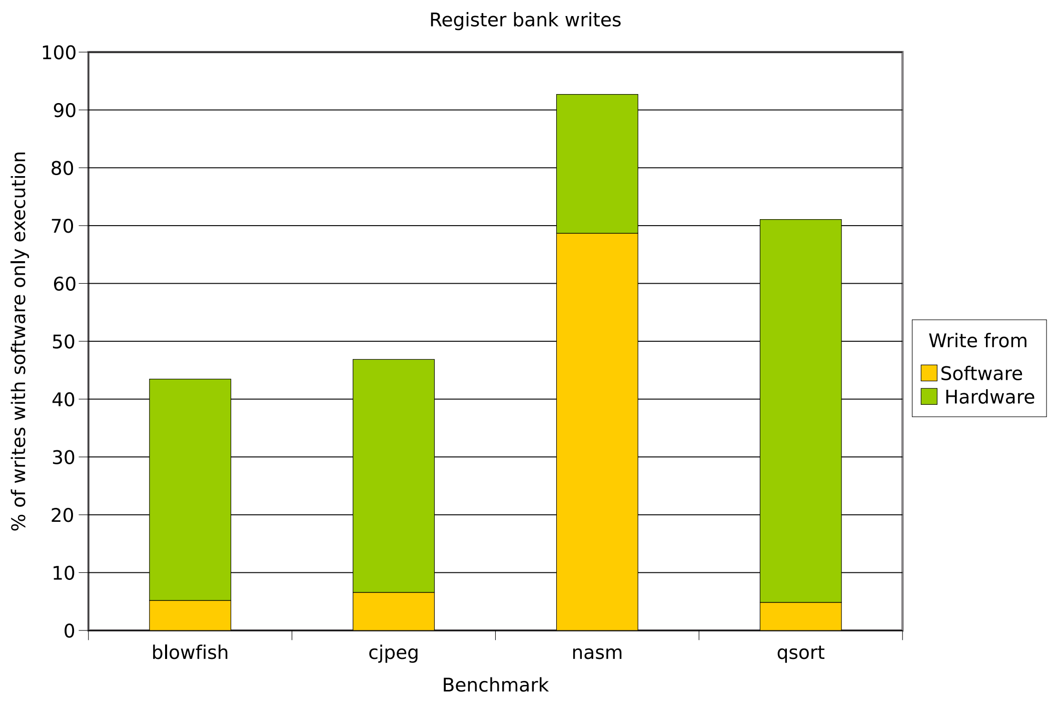


Figure 6.8: Remaining register bank writes

from instructions not detected to be in a hot spot and so continue to execute normally in software. Accesses from “hardware” are due to register reads to transfer values used in a block into the reconfigurable hardware and writes of valid values back into the register bank when the block exits.

Register reads

The remaining reads from software correspond approximately to the percentage of instructions that execute in software as shown in figure 6.1. Register bank reads from blocks executing hardware are reduced significantly over executing those blocks in software. This is due partly to values that are local to the block: these are held between operations in temporary storage locations within the reconfigurable hardware and not stored in and read from the register bank. Further reductions come from re-use of values in a block: these values are generally only read once from the register bank and then can be stored in the reconfigurable hardware for re-use[1]. In *cjpeg* and *blowfish* the number of register bank reads reduces to around 25%. In *nasm* the number of reads is reduced to just over 80%; this is mainly due to the relatively small proportion of instructions that execute in hardware in this benchmark.

Register writes

The remaining writes from software are approximately proportionally the same as for register reads; small variations are due only to slightly different proportions of register reads and writes in compiled and non-compiled blocks. The number of writes from hardware blocks is significantly higher than the number of remaining reads: approximately double the proportion of register writes remain in compiled blocks. This is due to the difficulty of determining

[1]A value may be read from the register bank multiple times if storing it within the block would be an impractical use of resources; for example if a value is only required at the start and end of a block. These results assume that a register value used in a block is only read once and stored in hardware for re-use.

whether values are "live" at the end of a block. In this system it is not known which values will be read by following instructions, therefore it must be assumed all registers written to during a block are "live" at the end. All registers written to during the block must have their final values written back. Only repeated writes to the same register within a block can be removed as these are temporary values local to the block.

The number of registers written back could be reduced by searching the following code paths after the block to determine which registers are live at the end of the block; however, this would require extra work to be performed when the block was compiled (see section 4.1.1).

Additionally procedure call standards could be used to reduce the number of registers written back. For example, the ARM procedure call standard defines registers R1-R3 to be scratch registers during a procedure call. If a block ended at a return these could be assumed to be invalid, removing the need to write values back to those registers. This requires the program to adhere to the procedure call standard.

Combining reads and writes, the total reduction in accesses is around 70% in *blowfish* and *cjpeg*, 50% in *qsort* and 15% in *nasm*.

6.3.3 Fetch and decode

The processes of fetching and decoding instructions also account for a significant proportion of the power consumption of microprocessors [MWea96]. The HCT is effectively a type of on-chip instruction cache where instructions have been translated to configure the hardware directly in a similar fashion to microcode. Instructions executed as part of a translated block will not need to be fetched from the instruction cache and decoded, reducing the energy consumption of these components. However, these savings will be offset in part by the operation of the HCT itself and the configuration of the hardware during block execution.

6.3.4 Hot spot detector and compilation process

The energy consumption of the hot spot detection hardware is expected to be small as the hot spot detector is only active during software execution: in all benchmarks except *nasm* this is for around 5% of instructions executed. The hot spot detector is only updated when a branch instruction is executed (about one in seven instructions) so it is only accessed in less than 1% of instructions executed in these benchmarks, ensuring low switching activity. BPT access in *nasm* will be more frequent, during around 10% of instructions, due to the larger proportion of instructions executed in software.

As with performance overhead, power overhead of compiling blocks is estimated to be small due to the low number of compilations (see figure 6.5). The energy consumption of performing a compilation would need to be very large, equivalent to executing many thousands of instructions, to have a significant overall impact.

6.4 Summary

The techniques described in this thesis can give an increase in performance of over double in some benchmarks. These improvements are due to introducing parallelism into hot spots and by reducing the number of operations performed. The system is best suited to code where a small number of hot spots dominate execution time, as is seen in *cjpeg* and *blowfish*.

There is also potential for reduced power consumption compared to superscalar microprocessors as the need to determine parallelism each time a section of code is executed is removed. A detailed power analysis of the techniques proposed in this thesis would be an interesting topic for future work in this area.

Chapter 7

Conclusions

This thesis presented a discussion on techniques for dynamically compiling program hot spots into reconfigurable hardware. An example system has been described and evaluated in terms of performance.

Chapter 3 presented a mechanism for rapid detection of hot spots based around a Block Profile Table containing counters for recently executed basic blocks. This operates as a cache of the most *hot* blocks and monitors how frequently they execute; these values are periodically decremented to bias recently executed basic blocks. This allows the BPT to be small: a table with 32-64 entries detects the majority of hot spots. The BPT also gathers information on the bias of the branch terminating the basic block; this is used to chain basic blocks linked by highly biased branches together.

A method of varying the sensitivity of the hot spot detector based on the number of active compiled hot spots was introduced. Variable sensitivity allows the hot spot detector to adapt to different program behaviour, preventing thrashing in programs with many hot spots, enabling less frequently executed blocks to be compiled in programs with few hot spots and allowing the set of compiled blocks to be replaced rapidly if behaviour changes during execution. This has been shown to adapt to different demands with different benchmarks.

Detected hot spots are compiled and stored in a Hardware Configuration Table. From here they are loaded into a reconfigurable hardware structure on demand. The size of this table determines the number of simultaneously available compiled hot spots. A small table of 32-64 entries allows 90-95% of instructions to be executed in hardware in programs that contain significant hot spots. Programs that do not contain significant hot spots require a larger table for the majority of the program to be executed in hardware; however, this comes at the cost of an increased number of compilations.

This hot spot detection mechanism is effective at detecting frequently executed basic blocks while only requiring a small amount of hardware. The hot spots constructed are small atomic entities with a single entry and exit point suitable for compiling into reconfigurable hardware. This is in contrast to other hot spot detection mechanisms which typically build long traces of instructions into the most frequently executed order.

Optimisations that can be performed on these detected hot spots were described in chapter 4. Using profile information a number of basic blocks can be chained together around highly biased branches to produce a larger atomic block of code for optimisation. These larger blocks allow a greater number of temporary values to be detected, reducing register bank accesses. Their atomic nature assumes a single path of control through the block, allowing internal control dependencies to be removed. This reduced number of dependencies allows increased parallelism and operations generating results only used by not-taken branches can be removed, yielding increased performance.

Chapter 5 described a suitable hardware execution engine for processing the data flow graphs generated by the processes described in the previous two chapters. The number of parallel functional units in the hardware was chosen to be four, based on an analysis of the available parallelism in compiled blocks and the potential performance gain versus decreased utilisation of functional units. Additional parallel hardware reduces utilisation; reduced hardware increases scheduling difficulty. It is likely that this structure is close to optimal within the constraints of compiling from sequential code and a single memory load per cycle. However, restrictions on the functional units that can perform certain operations, particularly memory stores, reduces scheduling flexibility and can reduce parallelism and therefore performance.

Such a structure can be compared to superscalar and VLIW machines, which typically execute around four parallel operations per cycle, due to inherent limits in instruction level parallelism of sequential code. Much of the parallelism available in algorithms cannot be extracted from sequential representations, hence the requirement of many reconfigurable computing systems to use different programming methods and to separate core parts of an algorithm to be parallelised statically.

Compiling from object code presents other difficulties in addition to extracting parallelism. Immediate operands pose such a problem. They are encoded into the instruction stream, which is not available to the reconfigurable execution engine. Storing them within the block configuration data would increase its size considerably. The solution proposed here is to access immediates from an *immediate pool*: this is configured at the start of each block with the values used in that block. Many immediate values, such as 1 and 0, are frequently re-used; this observation is exploited to reduce the number that must be stored in the block configuration data by preconfiguring the immediate pool to store these values at all times. However, this method relies on the majority of immediates to fall within this preconfigured set.

Finally, chapter 6 analysed the performance of the system as a whole. Programs that contain a small number of hot spots, such as image compression and encryption, approximately double in performance when compared to a scalar microprocessor due to increased parallelism. Performance in the *nasm* assembler only improves by around 15% as execution is spread over a large portion of the program and so a small number of hotspots do not dominate like in the other benchmarks, limiting the amount of regularly used code that can be stored in the HCT.

The compilation process has not been modelled in detail and so the performance impact of performing compilations can only be estimated. However, due to the small number of compilations this is expected to be significantly lower than the gain in performance caused by increased parallelism.

The improvements in performance are not as great as the improvements shown in some applications on reconfigurable coprocessor architectures such as PipeRench and Garp. However, in these systems core algorithms are selected statically, and require special design processes and tools to create the hardware configurations. In contrast, the system proposed in this thesis can provide these improvements while maintaining architectural compatibility with the original microprocessor, requiring no changes to programming methods, tools or existing object code. The level of parallelism extracted is instead comparable to superscalar or VLIW architectures. Again this system has advantages over VLIW in that existing code can be used; additionally the level of parallelism is not decided at compile-time as in VLIW, allowing future versions of the hardware to improve without requiring code to be recompiled. Superscalar also has these advantages; however, power consumption of superscalar architectures is high due to the requirement to schedule operations in parallel each time they are executed. The system described in this thesis only determines parallelism at block compile time: this information is then re-used when the compiled block is re-executed.

7.1 Limitations

The main limitation in this thesis is that the four benchmarks that were used in the analysis do not give a thorough evaluation of the system presented. While an effort was made to choose a variety of real-world applications, further examples are required to assess the effectiveness of the techniques presented in a wider variety of situations. The behaviour of *nasm* stands out in particular: this benchmark does not contain any significant hot spots and so performed relatively poorly compared to the other three benchmarks which all performed similarly.

Additionally, there are limitations in the compilation algorithms developed to map data flow graphs into the restrictive interconnect of the hardware structure. These do not take into account the detailed structure and limitations of the interconnect between functional units; therefore, the efficiency of scheduling into this structure is only an estimate based on hand-mapping of blocks into the structure.

7.2 Future work

This thesis has described an architecture for detecting hot spots and compiling them into a reconfigurable hardware structure at run-time. This has been evaluated in terms of performance; future work is required to evaluate the system in terms of energy consumption. In addition, the hot spots detected have limitations in that they cannot contain loops or multiple paths of control. Exploring the detection, compilation and mapping to hardware of such code structures presents an interesting area for future work.

7.2.1 Energy consumption evaluation

This system has only been evaluated in terms of performance compared to a scalar microprocessor. This system is designed to extract parallelism from a sequence of instructions at run-time; therefore, it would be interesting to evaluate this system against other architectural techniques that extract parallelism at run-time, such as superscalar architectures. This comparison should take into account both the performance and power consumption of the systems involved. The system described in this thesis has the potential for lower power consumption per unit of performance compared to a superscalar architecture as the process of detecting and extracting parallelism is only performed at block compile time, compared with each time the instructions are executed for a superscalar architecture.

This could be performed by modelling the proposed system in a hardware description language such as Verilog and then simulating it to measure the relative energy consumption of each component. This would not only give an accurate energy measurement, it could also be used to fine-tune the design to maximise energy savings.

7.2.2 Looping within compiled blocks

The system proposed can compile the body of a loop and execute it in hardware, but relies on software for the loop control. This is because allowing looping within hardware blocks removes the statically determinable properties of compiled blocks as the number of iterations cannot be determined at block compile time. This may cause overflows in the write buffer or load table. This restriction introduces overheads as values must be passed between loop iterations via the register bank and control is handed back to software at the end of each iteration. This also limits parallelism, as operations in different iterations cannot overlap.

Looping within blocks could be supported by introducing *checkpoints* into the block. These are points during the block where execution so far can be determined to be valid and so values in the block can be committed to registers and memory, overwriting the previous architectural state. If a breakout occurs, control is returned to software at the checkpoint instead of at the start of the block. Operations between checkpoints can be executed speculatively as before. Checkpoints would be inserted into each loop iteration. Loops that have been profiled to iterate many times could be assumed by the compiler to execute indefinitely. When the loop termination conditions do occur this would be treated as a breakout and state would be returned to the last checkpoint. The final iteration of the loop would then execute in software.

7.2.3 Multiple paths through compiled blocks

A further limitation in this system that execution is limited to a single path of control. This prevents small if-then-else structures being compiled into blocks. By allowing multiple conditional paths through a block these structures could be contained within compiled blocks, increasing block size and reducing software overheads. This would require support within the control of the hardware structure to use different parts of a block configuration depending on conditional tests within the block.

7.2.4 Compilation and scheduling algorithms

The algorithms required to schedule the data flow graphs produced by the hot spot detector into the reconfigurable hardware structure proposed have not been fully developed. This would have to be performed to verify the assumption made in this thesis that compilation can be performed at run-time without incurring significant overhead.

7.3 Summary

Run-time compilation of program hot spots into reconfigurable hardware can provide approximately double the performance of a scalar microprocessor when executing programs containing hot spots; a slight gain is achieved in less suitable programs. Superscalar and VLIW architectures provide similar performance benefits; however, this architecture is more flexible than VLIW as existing compiled code can be used, and has potential for lower power consumption than superscalar due to reduced work in determining parallelism. Further work in power analysis is required to determine the extent of any improvements in power consumption. The algorithms that perform well in this system are ones that perform a repeated set of operations on a large amount of data; this matches the behaviour of streaming media algorithms to process audio and video. This technique therefore has the potential to increase the capabilities of modern portable electronic devices, such as mobile phones and portable media players, which demand high performance in such applications.

References

[AFG$^+$00] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices*, 35(10):47–65, 2000.

[BDB00] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[BH03] M. Berndl and L. Hendren. Dynamic profiling and trace cache generation. In *CGO '03: Proc. International Symposium on Code Generation and Optimization*, pages 276–285, Washington, DC, USA, 2003. IEEE Computer Society.

[BR96] S. Brown and J. Rose. Architecture of FPGAs and CPLDs: A tutorial. *IEEE Design and Test of Computers*, 13(2):42–57, 1996.

[CCH$^+$00] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon. Stream computations organized for reconfigurable execution (score). In *FPL '00: Proc. The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, pages 605–614, 2000.

[CDN94] Andrea Capitanio, Nikil Dutt, and Alexandru Nicolau. Partitioning of variables for multiple-register-file VLIW architectures. In

ICPP '94: Proceedings of the 1994 International Conference on Parallel Processing, pages 298–301, 1994.

[CH97] A. Chernoff and R. Hookway. DIGITAL FX!32 — running 32-Bit x86 applications on Alpha NT. In *Proc. USENIX Windows NT Workshop, Seattle, Washington*, pages 9–13, 1997.

[CH02] K. Compton and S. Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, 2002.

[cjp] Independent JPEG Group. http://www.ijg.org.

[CLCG00] W. Chen, S. Lerner, R. Chaiken, and D. Gillies. Mojo: A dynamic optimization system. In *Proc. Third ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2000.

[CMH96] T. M. Conte, K. N. Menezes, and M. A. Hirsch. Accurate and practical profile-driven compilation using the profile buffer. In *Proc. 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-29)*, page 36, 1996.

[CPMC96] T. M. Conte, B. A. Patel, K. N. Menezes, and J. S. Cox. Hardware-based profiling: An effective technique for profile-driven optimization. *International Journal of Parallel Programming*, 24(2):187–206, 1996.

[CPSS00] Y. Chou, P. Pillai, H. Schmit, and J. P. Shen. PipeRench implementation of the instruction path coprocessor. In *Proc. International Symposium on Microarchitecture*, pages 147–158, 2000.

[CS00] Y. Chou and J. P. Shen. Instruction path coprocessors. In *Proc. 27th International Symposium on Computer Architecture*, 2000.

[DB00] E. Duesterwald and V. Bala. Software profiling for hot path prediction: less is more. *SIGPLAN Not.*, 35(11):202–211, 2000.

[DGB+03] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proc. International Symposium on Code Generation and Optimization*, pages 15–24, 2003.

[EA97] K. Ebcioglu and E. R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *ISCA*, pages 26–37, 1997.

[Ear03] R. Earnshaw. Procedure call standard for the ARM architecture. Technical report, ARM, October 2003.

[Eft02] A. Efthymiou. *Asynchronous techniques for power-adaptive processing*. PhD thesis, School of Computer Science, University of Manchester, 2002.

[EP00] A. J. Elbirt and C. Paar. An FPGA implementation and performance evaluation of the serpent block cipher. In *Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 33–40, 2000.

[Fis83] J. A. Fisher. Very long instruction word architectures and the ELI-512. In *ISCA '83: Proc. 10th Annual International Symposium on Computer Architecture*, pages 140–150, Los Alamitos, CA, USA, 1983. IEEE Computer Society Press.

[FS92] M. Franklin and G. Sohi. Register traffic analysis for streamlining inter-operation in fine-grain parallel processors. In *Proc. 25th Annual International Symposium on Microarchitecture*, pages 236–245, 1992.

[FS94] M. Franklin and M. Smotherman. A fill-unit approach to multiple instruction issue. In *MICRO 27: Proc. 27th Annual International Symposium on Microarchitecture*, pages 162–171, 1994.

[gcc] GCC, the GNU Compiler Collection. http://gcc.gnu.org.

[GHK$^+$91] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti. Building and using a highly parallel programmable logic array. *Computer*, 24(1):81–89, 1991.

[Gos96] G. R. Goslin. A guide to using field programmable gate arrays for application-specific digital signal processing performance. *Proc. SPIE*, 2914:321–331, 1996.

[GSM$^+$99] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. Piperench: a co/processor for streaming multimedia acceleration. In *ISCA '99: Proc. 26th Annual International Symposium on Computer Architecture*, pages 28–39, 1999.

[Har01] R. Hartenstein. Coarse grain reconfigurable architecture. In *ASP-DAC '01: Proc. 2001 conference on Asia South Pacific Design Automation*, pages 564–570, 2001.

[HFHK97] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera reconfigurable functional unit. In *Proc.IEEE Symposium on FPGAs for Custom Computing Machines*, pages 87–96, 1997.

[HKZ$^+$06] J. Hiser, N. Kumar, M. Zhao, S. Zhou, B. R. Childers, J. W. Davidson, and M. L. Soffa. Techniques and tools for dynamic optimization. In *Proc. IEEE International Parallel and Distributed Processing Symposium*, 2006.

[HM00] Z. Hu and M.Martonosi. Reducing register file power consumption by exploiting value lifetime characteristics. In *Proc. Workshop on Complexity-Effective Design*, 2000.

[HP] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, third edition.

[HSU$^+$01] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Rousseland. The microarchitecture of the Pentium® 4 processor. Intel Technology Journal, QI, 2001.

[HW97] J. R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, 1997.

[JS00] D. Jaggar and D. Seal. *ARM Architecture Reference Manual*. Addison Wesley Publishing Company, 2000.

[Kel75] R. M. Keller. Look-ahead processors. *ACM Computing Surveys*, 7(4):177–195, 1975.

[KGPK01] Gurhan Kucuk, Kanad Ghose, Dimitry V. Ponomarev, and Peter M. Kogge. Energy: efficient instruction dispatch buffer design for superscalar processors. In *ISLPED '01: Proc. 2001 International Symposium on Low Power Electronics and Design*, pages 237–242. ACM, 2001.

[Kla00] A. Klaiber. The technology behind Crusoe$^{\text{TM}}$ processors. Technical report, Transmeta Corporation, January 2000.

[Lev05] B. Levine. *HASTE: Hybrid architectures with a single, transformable executable*. PhD thesis, Carnegie Mellon University, May 2005.

[LL00] K. M. Lepak and M. H. Lipasti. Silent stores for free. In *International Symposium on Microarchitecture*, pages 22–31, 2000.

[MKG98] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Pipeline gating: speculation control for energy reduction. In *ISCA '98: Proc. 25th Annual International Symposium on Computer Architecture*, pages 132–141. IEEE Computer Society, 1998.

[MO99] T. Miyamori and K. Olukotun. REMARC: Reconfigurable multimedia array coprocessor. *IEICE Trans. Information Systems*, (2):389–397, 1999.

[MTG$^+$99] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *ISCA '99: Proc. 26th Annual International Symposium on Computer Architecture*, pages 136–147, 1999.

[MTL95] F. Mounes-Toussi and D. J. Lilja. Write buffer design for cache-coherent shared-memory multiprocessors. In *Proc. International Conference on Computer Design*, pages 506–511, 1995.

[MTN$^+$00] M. C. Merten, A. R. Trick, E. M. Nystrom, R. D. Barnes, and W. W. Hwu. A hardware mechanism for dynamic extraction and relayout of program hot spots. In *ISCA 2000: Proc. 27th Annual International Symposium on Computer Architecture*, pages 59–70, 2000.

[MVV$^+$03] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Proc. Field-Programmable Logic and Applications*, pages 61–70, 2003.

[MWea96] J. Montanaro, R.T. Wite, and K. Anne et al. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *IEEE Journal of Solid-State Circuits*, 31:1703–1714, 1996.

[nas] The Netwide Assembler: NASM. http://nasm.sourceforge.net.

[NH97] R. Nair and M. E. Hopkins. Exploiting instruction level parallelism in processors by caching scheduled groups. In *International Symposium on Computer Architecture*, pages 13–25, 1997.

[NZ04] A. Niyonkuru and H. C. Zeidler. Designing a runtime reconfigurable processor for general purpose applications. In *Proc. 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, pages 143–149, 2004.

[Pag94] I. Page. The HARP reconfigurable computing system. Technical report, Oxford University Hardware Compilation Group, 1994.

[PL01] S. Patel and S. Lumetta. rePLay: a hardware framework for dynamic optimization. *IEEE Transactions on Computers*, 50(6), 2001.

[RBS96] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *International Symposium on Microarchitecture*, pages 24–35, 1996.

[RLG$^+$98] C. R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. M. Arnold, and M. Gokhale. The NAPA adaptive processing architecture. In *FCCM '98: Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, page 28, 1998.

[RS94] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proc. 27th Annual International Symposium on Microarchitecture*, pages 172–80, 1994.

[Sch93] B. Schneier. Description of a new variable-length key, 64-bit block cipher (Blowfish). In *Fast Software Encryption, Cambridge Security Workshop Proceedings*, pages 191–204, 1993.

[SLL$^+$00] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–481, 2000.

[SR00] M. S. Schlansker and B. R. Rau. EPIC: Explicitly parallel instruction computing. *Computer*, 33(2):37–45, 2000.

[Val05] M. G. Valluri. *A Hybrid-Scheduling Approach for Energy-Efficient Superscalar Processors*. PhD thesis, The University of Texas at Austin, 2005.

[WC96] R. Wittig and P. Chow. OneChip: An FPGA processor with reconfigurable logic. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 126–135, Los Alamitos, CA, 1996. IEEE Computer Society Press.

[WH95] M. J. Wirthlin and B. L. Hutchings. DISC: the dynamic instruction set computer. In *Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, Proc. SPIE 2607*, pages 92–103, 1995.

[ZPG$^+$00] H. Zhang, V. Prabhu, V. George, M. Wan, M. Benes, A. Abnous, and J. Rabaey. A 1-V heterogeneous reconfigurable DSP IC for wireless baseband digital signal processing. *IEEE Journal of Solid-State Circuits*, 35(11):1697–1704, 2000.