Technical Report

UCAM-CL-TR-707
ISSN 1476-2986

Number 707

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Complexity-effective superscalar embedded processors using instruction-level distributed processing

Ian Caulfield

December 2007

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

© 2007 Ian Caulfield

This technical report is based on a dissertation submitted
May 2007 by the author for the degree of Doctor of
Philosophy to the University of Cambridge, Queens' College.

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

Modern trends in mobile and embedded devices require ever increasing levels of performance, while maintaining low power consumption and silicon area usage. This thesis presents a new architecture for a high-performance embedded processor, based upon the instruction-level distributed processing (ILDP) methodology. A qualitative analysis of the complexity of an ILDP implementation as compared to both a typical scalar RISC CPU and a superscalar design is provided, which shows that the ILDP architecture eliminates or greatly reduces the size of a number of structures present in a superscalar architecture, allowing its complexity and power consumption to compare favourably with a simple scalar design.

The performance of an implementation of the ILDP architecture is compared to some typical processors used in high-performance embedded systems. The effect on performance of a number of the architectural parameters is analysed, showing that many of the parallel structures used within the processor can be scaled to provide less parallelism with little cost to the overall performance. In particular, the size of the register file can be greatly reduced with little average effect on performance – a size of 32 registers, with 16 visible in the instruction set, is shown to provide a good trade-off between area/power and performance.

Several novel developments to the ILDP architecture are then described and analysed. Firstly, a scheme to halve the number of processing elements and thus greatly reduce silicon area and power consumption is outlined but proves to result in a 12–14% drop in performance. Secondly, a method to reduce the area and power requirements of the memory logic in the architecture is presented which can achieve similar performance to the original architecture with a large reduction in area and power requirements or, at an increased area/power cost, can improve performance by approximately 24%. Finally, a new organisation for the register file is proposed, which reduces the silicon area used by the register file by approximately three-quarters and allows even greater power savings, especially in the case where processing elements are power gated.

Overall, it is shown that the ILDP methodology is a viable approach for future embedded system design, and several new variants on the architecture are contributed. Several areas of useful future research are highlighted, especially with respect to compiler design for the ILDP paradigm.

Acknowledgements

I would like to thank my supervisor, Simon Moore, for his advice, guidance and most particularly his patient support over the past few years. Thanks are also due to James Srinivasan and Simon 'Harry' Hollis (and to a lesser degree many other members of the Computer Architecture Group) for fielding innumerable queries along the lines of "Do you know anything about . . . ?".

I would like to give special recognition to Diana Wood for providing unconditional support and putting up with me while I completed my thesis, and for providing assistance with proofreading. Unthanks go to the members of Cambridge University Bowmen who, whilst providing many hours of friendship and recreation, conspired to lengthen my stay at university!

My research was funded with a grant from the Engineering and Physical Sciences Research Council.

Contents

1 Introduction **15**
1.1 Motivation . . . 15
1.2 Contribution . . . 16
1.3 Outline . . . 16

2 Background **17**
2.1 Parallelism . . . 17
2.2 Control-flow vs data-flow . . . 18
2.2.1 Static data-flow . . . 19
2.2.2 Coloured data-flow . . . 19
2.2.3 Tagged-token data-flow . . . 19
2.3 Pipelining . . . 20
2.3.1 Data hazards . . . 21
2.3.2 Control hazards . . . 21
2.4 Superscalar architectures . . . 22
2.4.1 In-order superscalar . . . 22
2.4.2 Out-of-order superscalar . . . 23
2.4.3 Parallels between out-of-order and data-flow . . . 25
2.5 Multithreading . . . 25
2.5.1 Simultaneous multithreading . . . 25
2.6 Vector architectures . . . 26
2.7 Decoupled architectures . . . 26
2.8 Dependence-based architectures . . . 27
2.9 Instruction-level distributed processing . . . 28
2.10 Multiprocessing . . . 28
2.10.1 Symmetric multiprocessing . . . 29
2.10.2 Non-uniform memory architectures . . . 29
2.10.3 Multicore processors . . . 30
2.11 Instruction encoding . . . 30
2.11.1 CISC . . . 30
2.11.2 RISC . . . 31
2.12 Embedded system design . . . 31
2.12.1 Low cost . . . 32
2.12.2 Low power consumption . . . 33

2.12.3 Real-time performance 35
2.13 Future developments 35

3 ILDP in an Embedded Context **37**
3.1 Details of ILDP 37
3.2 Instruction set 39
3.3 Programming model 40
3.3.1 Register assignments 40
3.3.2 Memory organisation 42
3.3.3 Function call specification 42
3.4 Microarchitecture 43
3.5 Complexity 46
3.6 Complexity analysis 47
3.6.1 Register rename logic 47
3.6.2 Issue window dependency logic 48
3.6.3 Instruction issue logic 49
3.6.4 Register file 49
3.6.5 Bypass paths 50
3.6.6 Data caches 50
3.7 Power consumption 51
3.8 Summary 52

4 Evaluating ILDP **53**
4.1 Performance vs other embedded architectures 53
4.1.1 Methodology 53
4.1.2 Compiler toolchain 53
4.1.3 Benchmarks 55
4.1.4 Simulation 55
4.1.5 Results 56
4.1.6 Conclusion 60
4.2 Resource utilization 60
4.2.1 Decode / Issue logic 60
4.2.2 Processing elements 61
4.2.3 Register rename logic 62
4.3 Parameter space exploration 64
4.3.1 Issue width 64
4.3.2 Rename bandwidth 67
4.3.3 FIFO depth 69
4.3.4 Register network bandwidth 73
4.3.5 Physical register file size 76
4.3.6 ISA register file size 78
4.3.7 Cache parameters 80
4.4 Combining approaches 80
4.5 Summary 82

5 Architectural developments **85**
5.1 Multiplexing instruction FIFOs onto a processing element 85
5.1.1 Implementation details 86
5.1.2 Evaluation 87
5.1.3 Conclusions 87
5.2 Combining memory access units 90
5.2.1 ISA modifications 91
5.2.2 Implementation details 92
5.2.3 Evaluation 93
5.2.4 Further developments 101
5.2.5 Implementation details 101
5.2.6 Evaluation 103
5.2.7 Cache structure 107
5.2.8 Conclusions 107
5.3 Register inlining 109
5.3.1 Implementation details 110
5.3.2 Complexity 112
5.3.3 Evaluation 112
5.3.4 Conclusions 113
5.4 Summary 114

6 Conclusions **115**
6.1 Summary 115
6.2 Conclusion 116
6.3 Future work 117

Bibliography **121**

List of Tables

3.1 The ILDP instruction set 41
3.2 Register assignments for the ILDP ABI 42

4.1 Architectural parameters for simulation 57

5.1 Instruction set modifications 91

List of Figures

3.1 Ways of grouping instructions by their dependencies 38
3.2 Instruction encodings 40
3.3 Function stack frame layout 43
3.4 The ILDP architecture, as outlined by Kim and Smith 44
3.5 Structure of an ILDP processing element 45

4.1 Performance of ILDP compared to embedded processors 58
4.2 Number of instructions executed, compared to ARM 58
4.3 IPC of tested processors 59
4.4 Number of instructions processed by the parcel, rename and steer pipeline stages 61
4.5 Processing element activity 62
4.6 Distribution of rename map bandwidth usage over time 63
4.7 Performance of ILDP at various issue widths, normalised to 4-way case 65
4.8 Number of instructions in the parcel, rename and steer pipeline stages 66
4.9 Effect on performance of removing register rename map ports 67
4.10 Issue pipeline statistics when varying number of rename map ports 68
4.11 Distribution of instruction strand lengths in ILDP code 70
4.12 Instruction FIFO utilization 71
4.13 Performance at various instruction FIFO sizes, compared to an unbounded FIFO 72
4.14 Register network bandwidth utilization 74
4.15 Performance at various register network bandwidths, compared to unbounded case 75
4.16 Effect of scaling physical register file size from 128 GPRs on performance . . . 77
4.17 Effect of scaling both physical and logical register file size on performance . . 79
4.18 Effect of increasing cache associativity from 2-way on performance 81
4.19 Effect of varying several architectural parameters 82

5.1 Multiplexing instruction FIFOs between processing elements 86
5.2 Effect on performance of multiplexing instruction FIFOs between PEs 88
5.3 Processing element utilization with two FIFOs per PE 89
5.4 Design of memory access unit 91
5.5 Additional instruction encodings 92
5.6 Performance of the MAU architecture variant compared to the baseline 94
5.7 Performance of the MAU variant when multiple outstanding loads are implemented 96

5.8 Effect of reducing register rename bandwidth on performance in the MAU model 97
5.9 MAU FIFO occupancy by time 99
5.10 Performance at various MAU FIFO sizes, compared to an unbounded FIFO . . 100
5.11 Design of memory access unit, using out-of-order issue 102
5.12 Performance of various models of out-of-order memory access unit 104
5.13 Effect of out-of-order MAU instruction window size on performance 106
5.14 Effect of increasing L1 cache size on performance (compared to 8KB size) . . . 108
5.15 Register tag/value portion of PE instruction FIFO 111

6.1 Traditional CPU arrangements for multiprocessing 118
6.2 An example four-way multiprocessing ILDP CPU 119

1 Introduction

This dissertation describes a number of architectural designs aimed toward producing high-performance embedded microprocessors. These build upon the techniques of instruction-level distributed processing developed by Kim and Smith [30]. I show that these techniques allow processors to achieve high performance without many of the costs in power consumption and circuit complexity associated with traditional superscalar architectures.

1.1 Motivation

The embedded microprocessor market is currently growing at a greater rate than the market for desktop and server processors. More and more products in various market sectors are being produced with embedded processors – a modern car can contain dozens of embedded processors controlling various different features within the vehicle. While the desktop microprocessor market has the highest profit margins, the embedded processor market ships far higher volumes per year, yielding large overall profits. With large growth and revenues, and increasing demands for greater functionality at a lower cost, there is considerable scope for research into processor architecture targeted at embedded systems.

In the mobile device sector, there is an increasing trend toward 'all-in-one' devices that perform a number of functions: a modern smart phone will be able to connect to several varieties of wireless network, play back (and even record) audio and video, play games, take photos and browse the Internet. In order to perform these functions, the device must be capable of running one or more radio protocol stacks, perform audio and video encoding and decoding, render graphics in real time, perform image processing and be able to execute external scripts or applications. While a single-function device can be constructed using a simple low-performance microprocessor and some special-purpose logic, a device with such diverse usage requirements but similar power and size constraints will generally require a high-performance general purpose processor, which can easily shift from one application to another. Many of the techniques used to achieve high performance in desktop processors are unsuitable for embedded systems due to their power or area requirements, and so alternative approaches are required.

1.2 Contribution

In this thesis, I will show that instruction-level distributed processing provides an approach to designing low-complexity microprocessors providing high performance with a smaller area and power penalty than an out-of-order superscalar approach. The ILDP model is evaluated in depth, and several parameters which are likely to have a large effect on the complexity of the design are explored to find the optimum trade-off between complexity and performance. Several new developments on the architecture are proposed in order to reduce the overall design area, and thus power consumption.

1.3 Outline

The remainder of this dissertation is structured as follows:

Chapter 2 introduces and describes many of the issues present with and techniques used in modern microprocessor design, particularly looking at exploiting parallelism for high performance and the challenges of designing for embedded systems.

Chapter 3 describes the instruction-level distributed processor methodology and architecture on which this work is based, and outlines the implementation built upon them. The complexity of this design is analysed, compared to typical processor architectures.

Chapter 4 analyses the performance of the ILDP design and investigate the effect various architectural parameters have on performance and complexity.

Chapter 5 describes several new designs built upon the ILDP architecture intended to reduce circuit complexity and power consumption and make this approach more suitable for embedded implementation while maintaining high performance.

Finally Chapter 6 summarises this work and presents conclusions drawn. The contributions made by this work are outlined and possible areas for future work are described.

Background 2

Processor performance can be reduced to the following equation [48]:

$$T_{exe} = \frac{n_{inst}}{F_{clk} \times IPC} + T_{mem} \tag{2.1}$$

where T_{exe} gives the overall time to execute a program or set of programs, F_{clk} refers to the clock frequency of the system, n_{inst} is the total number of instructions executed, IPC is the average number of instructions executed within a single clock cycle and T_{mem} is the time spent by the system waiting for external activities such as I/O or memory accesses. Performance is greater when T_{exe} is smaller.

Performance can thus be increased by maximising F_{clk} and IPC and minimising n_{inst} and T_{mem}. Clock frequency can be increased by technology scaling as smaller manufacturing processes become available, by pipelining or by redesigning or replacing complex logic blocks in order to reduce the critical path. IPC can be increased by exploiting more parallelism within an architecture. The total instruction count can be reduced by optimising the ISA or improving the compiler used. Memory access time is generally reduced through the use of caches and speculatively issuing memory instructions as early as possible.

Technology scaling is creating a trend whereby the delays due to logic are becoming less expensive and those due to wiring are becoming more expensive, since as feature sizes decrease, overall die sizes tend to remain the same with additional logic added. A cross-chip wire remains the same length, but the delay to drive a signal along it increases [28] and as clock speeds increase the number of clock cycles required to drive a signal across a chip rises drastically. When scaling smaller structures, the transistor delay will decrease at a greater rate than the wire delay, which means that circuit structures with long wire delays thus scale badly as feature sizes decrease. Future architectures must be able to localise data usage and minimise, or tolerate delays in, global communication.

2.1 Parallelism

In order to increase IPC, an architecture must be able to exploit parallelism present within the workload it is executing. An important thing to note is that a workload with little or no inherent

parallelism will benefit very little from a parallel architecture – a single program that consists of a linear sequence of dependent instructions is likely to execute just as well on a simple scalar architecture as a complex, multithreaded, superscalar one and may actually perform better on the scalar processor due to the lower overhead of the less complex logic. There are two main forms of parallelism that can be exploited in hardware:

1) Instruction-level parallelism (ILP) – in general, when taking a group of instructions from a program, while there will be many interdependencies between them, there will be sets of instructions that are mutually independent and thus can be executed in parallel. A special case of ILP is *loop-level parallelism* (LLP) – where successive iterations of a loop operate on independent data and can therefore be executed in parallel. Such loops are parallelized to extract ILP through a process called *loop unrolling*.

2) Thread-level parallelism (TLP), where more than one program (or *thread*) can be executed in parallel. Since programs generally communicate via main memory, if there is some mechanism for synchronizing memory accesses then all instructions from separate programs are independent and can be executed concurrently. Processors that take advantage of TLP can make up for a shortfall in ILP, but rely on there being more than one thread of execution in order to make any gains. A processor that can execute more than one thread concurrently can help reduce T_{mem}, as when one thread stalls on a memory access, another can execute in its place. TLP workloads can be further characterised based on how tightly coupled the threads of execution are; a highly coupled workload will require a lot of communication and synchronization between threads, whereas a set of completely independent programs will be very loosely coupled. Depending on the way in which an architecture exploits TLP, it may perform better for more or less coupled workloads.

2.2 Control-flow vs data-flow

Most microprocessors use a *control-flow* architecture, where a program is represented as a linear sequence of instructions. The model processor executes instructions in order, using a program counter (PC) to keep track of the current location in the program. Instructions to modify the PC are provided in order to implement conditional expressions, loops and subroutines. As programs are represented as a linear sequence, exploiting ILP is difficult, since dependencies between instructions must be resolved in order to allow non-linear execution. Also, since the instruction scheduling is determined statically by the compiler, the processor cannot easily adapt dynamically to the execution environment – for example, long latency operations such as cache misses are likely to cause a processor stall even if other operations could theoretically be executed.

Data-flow architectures represent programs as a data-flow graph: each node in the graph is an instruction, and each arc is a data dependency between instructions. There is no explicit

ordering of the instructions beyond that of the dependencies – any instruction whose source operands are available can be executed at any time, making ILP easier to exploit. If a single instruction stalls (for example due to a cache miss) other operations may be issued instead while data for them is available. A *matching store* is used to find nodes available for execution and issue them to functional units.

2.2.1 Static data-flow

In the static data-flow model each arc can have at most one datum (or *token*) on it at any one time. A node becomes available for execution when there are tokens on all of its input arcs. A reverse signalling mechanism is used for flow control – when a node executes, it signals to all its predecessors that it has consumed its input tokens and that they may place new tokens on the arcs.

Due to the single token per arc limitation, this model has problems implementing shared functions as multiple instances of the same function cannot execute concurrently, and there is usually a limit on the number of backward signalling arcs. These kinds of functions generally need to be implemented by replicating the function body, which is inefficient.

2.2.2 Coloured data-flow

With coloured data-flow, each arc may carry more than one token. Each token is assigned a *colour* and nodes that take more than one input will only match tokens with the same colour. Result tokens are given the same colour as their source tokens. A function call generates a new colour for its tokens, allowing concurrent calls to the same function, as the different colours for each invocation will prevent conflicts.

The circuitry required for colour-matching is complicated, expensive and difficult to pipeline, which can cause performance problems with this model.

2.2.3 Tagged-token data-flow

The tagged-token dynamic model moves the token storage into *activation frames* – each function invocation creates an activation frame, similar to stack frames in a control-flow processor. When the first token arrives for a dyadic operation, its data is stored in the activation frame. When the second token arrives, the data from the first token is retrieved from the activation frame and the operation executes.

This approach is much simpler to implement than the coloured dynamic model and can be pipelined.

2.3 Pipelining

Several different pieces of hardware are required in a processor – for example: a load unit to fetch the current instruction word from memory; logic to interpret the instruction's opcode and generate control signals for successive hardware blocks; a register file to hold register values; an ALU to perform computation and a memory unit to load and store data values from memory. When an instruction is executed, that instruction must generally pass through these blocks in sequence, and only one block will be active at one time. *Pipelining* is a technique that allows higher performance by allowing multiple instructions to be passing through this sequence of hardware blocks, known as the 'pipeline'. Latches are placed between each block to synchronize the data movement to the clock. A classic example is the standard RISC pipeline, split into five stages:

- Instruction fetch
- Instruction decode and register access
- Execute
- Memory access
- Register write-back

A pipeline of n stages increases the typical instruction latency from 1 to n clock cycles, but reduces the cycle time by nearly n and allows (potentially) n instructions to be executed concurrently. Theoretically, this means that the frequency of the processor is increased by a factor of n while maintaining an optimum IPC of 1. In practice the latches add extra circuit delay and not all hardware blocks necessarily have equal delay – the clock period must be set to the worst-case delay of the slowest block, plus the overhead of the latches. If one pipeline stage has a delay much longer than the others, then it can significantly limit performance.

Pipelining can introduce new issues, however, which are generally referred to as *hazards*. These are generally divided into *data hazards*, where problems arise with the transfer or validity of data values, and *control hazards*, where problems arise with the flow of execution of a program.

2.3.1 Data hazards

In the case of the 5-stage RISC pipeline, if an instruction depends upon a register value generated by its immediate predecessor, then when it comes to read the register value during the 'register access' pipeline stage it will incorrectly read the previous value of the register, as the predecessor will still be in the 'execute' stage and will not have written its result back to the register file.

Data hazards are generally resolved using *bypass* (or *forwarding*) data paths. For the 5-stage RISC pipeline, the results of the 'execute' and 'memory access' pipeline stages are fed directly into the execute stage on the next cycle, bypassing the register file entirely. The decode logic then uses a technique known as *scoreboarding* to keep track of which registers' values should be read from the register file and which are currently in the pipeline and should use the bypass paths.

However, not all data hazards may be resolved like this – if a load instruction is immediately followed by an instruction using the result of the load, the second instruction will enter the 'execute' stage at the same time as the load instruction is in the 'memory access' stage, performing the load. With this pipeline arrangement it is impossible to execute these instructions simultaneously, as the second instruction requires the result of the load at the time the load commences. As such, it is impossible to resolve this hazard without delaying the second instruction. Often it is possible for the compiler to re-order instructions to remove this hazard, but when this is not possible there are two basic strategies for dealing with this situation in the processor:

- Software interlocking
  The processor does nothing to resolve this conflict – it is entirely up to the compiler to ensure this situation does not occur, and to insert no-ops into the instruction stream where necessary.
- Hardware interlocking
  The processor uses scoreboarding to determine when these hazards will arise, and inserts no-ops as necessary into the pipeline itself.

2.3.2 Control hazards

When an unconditional branch is decoded, the instruction following it in memory will already be in the process of being fetched from memory. The processor can either choose to execute this instruction (the ISA would then specify that the instruction immediately following a jump is always executed) which is then termed as being in a *branch delay slot*, or it can nullify it and allow a no-op to proceed down the pipeline. The advantage of using branch delay slots is that it simplifies issue logic and can allow slightly better performance. However, branch delay slots

can cause issues with code compatibility, as the the number of delay slots following a branch depends on the pipeline organisation – a new version of an architecture may require a different number of delay slots than a previous architecture yet still have to run code compiled for the old architecture. The new processor would then have to emulate a different number of delay slots, negating the original issue logic complexity advantages.

A further problem arises when a conditional branch is executed – if the branch condition is resolved in the 'execute' pipeline stage then the new PC value will not be available until several cycles after the branch was fetched, which can be a large penalty if the architecture uses a deep pipeline. This problem is generally tackled using *branch prediction*. In the simplest case, all conditional branches can be assumed to be not taken, meaning that the instruction fetch stage continues fetching instructions as it would if the branch had not come up – this means there is a performance penalty for a taken branch, but none for one that is not taken. More sophisticated schemes include static branch prediction, where the branch opcode specifies whether it should be predicted as taken or not taken, and dynamic branch prediction, where the processor maintains some state on past behaviour of branches in order to guess whether a branch will be taken or not. Modern dynamic branch predictors can achieve an accuracy in excess of 99%.

2.4 Superscalar architectures

Superscalar architectures (as opposed to standard, or *scalar* architectures) increase performance by executing multiple instructions in parallel, thereby increasing IPC. Superscalar architectures are generally classified as either being *in-order* or *out-of-order*.

2.4.1 In-order superscalar

In-order architectures are so named because they can only issue instructions in the order they appear in the instruction stream. For a processor of issue width n, up to n instructions can be issued simultaneously, but only if all n instructions are independent. An instruction cannot be issued in the same cycle as one it depends upon, and none of the instructions following it can be issued either. For this reason, in-order architectures require good compiler support in order to arrange the instructions in the program in such a way as to be able to maximise parallelism. Load stalls and other long latency operations can be a problem, as no further instructions can be issued while the processor is waiting for the data to come back from memory, regardless of whether they depend on the result or not. Some architectures provide *speculative load* (or *pre-fetch*) instructions which allow a compiler to insert hints as to future load instructions earlier in the instruction stream, ahead of conditional operations – any exceptions caused by the speculative load are ignored, as the actual load might not be executed. This allows load data to

be brought into the cache ahead of the instructions that require it, potentially eliminating some stalls due to cache misses.

In-order architectures can be further divided depending on whether or not they support *out-of-order completion*. Even when instructions are issued strictly in order, if different instructions have different latencies, they can complete out of order (this is particularly true with memory access or floating point instructions). This issue can also arise on single-issue processors if instructions are allowed to issue on the integer pipeline while slower floating-point instructions are still executing. If a processor does not support out-of-order completion, then it will stall the pipeline when an instruction has a high latency, blocking later instructions from finishing first.

Out-of-order completion introduces additional problems, for example maintaining a consistent state becomes difficult if an instruction causes an exception after some of its successors have already executed. In some cases it is possible to allow this using *imprecise exceptions* – for instance, in the case of an arithmetic exception, if the process will be terminated rather than restarted, it may not be necessary to preserve a consistent state. Some architectures provide 'checkpoint' instructions, where any exceptions before the checkpoint must be resolved before any effects of later instructions become visible. However there are many cases where precise exception handling is required, e.g. branch mispredictions or memory page faults to pages that have been swapped out. In these cases architectures generally use a mechanism known as *in-order commit* – while instructions can execute out-of-order and propagate their results to other instructions, their results do not become visible in the architectural state (i.e. the register file and system memory) until they commit, a process that happens strictly in-order. In the case of an exception, any uncommitted instructions following the instruction causing the exception are flushed and the system reverts to the last committed state at the exception point before executing any exception handlers. Smith describes two approaches to implementing in-order commit in [51]: the *reorder buffer*, where instruction results are stored until they can be committed to the register file; or the *history buffer*, which holds old values of registers until they are no longer needed, from which the register file is restored on an exception.

2.4.2 Out-of-order superscalar

Out-of-order architectures [29; 52] can scan a 'window' of instructions following the current committed execution point, and can execute any instruction from the window whose input operands are available. These architectures avoid several of the bottlenecks present in in-order architectures – a load instruction can be executed as soon as its address is known, rather than waiting until all previous instructions have executed, and if a load stalls successive instructions can continue executing if they do not depend on the load result.

In order to maintain a sufficient level of parallel execution, large structures are necessary to analyse data dependencies and identify available instructions. These hardware structures consume a lot of power, making them less suitable for embedded systems.

With large instruction windows a problem arises in that there are *false dependencies* between instructions. In the case of the instruction sequence:

```
R3 := R3 * R5
R4 := R3 + 1
R3 := R5 + 1
R7 := R3 * R4
```

In this sequence, the fourth instruction depends upon the second and third and the second depends upon the first – however, since all four instructions use the register R3, the third will be identified as depending upon the second (the second instruction must read R3 before the third writes to it). The problem is particularly acute for looping code, as each iteration of the loop uses the same registers, preventing any possible parallelization of the loop.

This problem is generally solved using one of two different approaches: using *register renaming* or a *reorder buffer*.

With register renaming the processor has a register file containing more registers than the ISA specifies, and any instruction that writes to a register is allocated a new register from the expanded file. The processor maintains a *map table* which identifies which of the actual registers are currently holding the values of the registers specified by the ISA. Since the first and third instructions in the sequence above will be allocated distinct registers, the dependency between them will be broken and the third instruction can be executed in parallel with the first two. In the case of loops, each iteration of the loop can be allocated a fresh set of registers, so a parallelizable loop can be unrolled and executed in parallel. Since most instructions in typical code write to a register, the size of the register file must be similar to that of the instruction window.

A reorder buffer holds the data from speculatively executed instructions. The register file contains only those values from committed instructions – on a rollback, the register file contains the consistent architectural state. When an instruction executes, it writes its result back to the reorder buffer, then instructions commit in order from the reorder buffer and write back to the register file. An issuing instruction can read data from either the register file or the reorder buffer, and can issue when all its operands are available from either source. The *register update unit* [55] is a development on the reorder buffer which handles instruction issue, dependency checking and in-order commit in a single structure.

Reorder-buffer-based architectures have the advantage that the difficulties of maintaining precise architectural state are resolved along with allowing out-of-order issue, but generally require content-addressable memory in their implementation, and thus do not scale well. Register renaming scales better, but it is necessary to be able to restore previous versions of the map table in the event of a rollback.

To fully utilize superscalar architectures, sufficient instruction-level parallelism (ILP) must be present within the executed code. Smith [54] and Wall [61] explore the limits of such parallelism within typical processor workloads.

2.4.3 Parallels between out-of-order and data-flow

Modern superscalar architectures effectively map control-flow input onto a data-flow execution engine – the reorder buffer acts as a matching store over a static data-flow program. The single-token-per-arc limitation of static data-flow is resolved using register renaming and loop unrolling – an instruction that is executed multiple times concurrently is replicated with the reorder buffer, with different register numbers. The size of the instruction window limits the amount of possible parallelism as it restricts the size of the generated data-flow program.

2.5 Multithreading

Multithreaded architectures allow multiple execution threads to run on a processor concurrently. The processor is augmented with multiple 'hardware contexts' – consisting of the register file, program counter and any other per-thread architectural state – and can execute instructions from any of the running threads, exploiting TLP. This allows T_{mem} to be decreased, as time spent waiting for memory access can be used to execute another thread, effectively masking memory latencies.

Multithreaded processors can provide performance improvements on a level comparable to superscalar architectures at a much reduced power usage, making them attractive options for embedded systems, given a sufficiently threaded workload [62; 63]. Since multithreaded architectures share processor resources (in particular the memory hierarchy), they perform well with tightly-coupled TLP workloads.

2.5.1 Simultaneous multithreading

Simultaneous multithreading (SMT) [10; 37; 59] architectures combine superscalar execution with multithreaded instruction fetching. Instructions from multiple threads can be issued and executed simultaneously, improving the overall *IPC* achieved.

Most studies into SMT architectures use out-of-order superscalar techniques, to further increase *IPC*. The two technologies complement each other well, as typical workloads tend to be quite 'bursty' in the amount of ILP that can be extracted – the average utilization of the functional

units within the processor tends to be quite low, but they are necessary in order to provide high peak performance. An SMT architecture allows instructions from multiple threads to fill in the gaps.

Several high-end processors are now available which support multithreading – some models of the Intel Pentium 4 and Core 2 Extreme processors implement 'Hyperthreading' [38], the MIPS 34K core implements 'MIPS MT' [39] and the IBM POWER5 CPU can execute two threads on each core.

While SMT architectures allow higher utilization of processor resources, thereby increasing performance at a relatively small complexity cost, they can also greatly increase the requirements on the memory hierarchy, requiring highly associative caches to maintain performance [4; 15; 21; 22].

2.6 Vector architectures

While superscalar processors exploit parallelism by executing multiple instructions from a scalar instruction stream concurrently, vector architectures provide instructions that operate on vectors of data. A scalar add instruction will produce the sum of two values, while a vector add might sum eight pairs of values at once. This technique is known as Single Instruction, Multiple Data (SIMD) and generally exploits loop-level parallelism – if a loop operates on an array of data, where iterations of the loop do not depend on each other's results, then SIMD can be used to execute several iterations of the loop simultaneously. The additional hardware requirements for SIMD are less than those for superscalar processing – the instruction decode and issue logic remains the same, but register and ALU width are increased, using more silicon area but without greatly increasing complexity.

Many modern CPU architectures include SIMD instructions in their ISA (e.g. Intel's MMX and SSE, AMD's 3DNow, ARM's NEON) and combine SIMD with superscalar for additional potential performance gains. SIMD support in compilers has lagged behind the hardware availability due to the difficulty of parallelizing linear code – many compilers are unable to generate SIMD code automatically, instead relying on the programmer to vectorize the programs manually.

2.7 Decoupled architectures

Decoupled architectures [47] separate an instruction stream into two streams – one to access memory and the other to perform computation. This allows the memory, or 'access', stream to

run ahead of the ‘execute’ stream, and so helps to reduce T_{mem}. Decoupled techniques were used in the CRAY-I supercomputer, as well as more recent projects such as PIPE [12] and MISC [60].

Krashinsky and Sung [34] investigate the possibilities of using decoupled architectures to reduce complexity in general-purpose processors, while Talla and John [57] look specifically at multimedia applications.

Kudriavtsev and Kogge [35], Sung et al. [56] describe methods of combining SMT techniques with decoupled architectures.

Crago et al. [7] extend the basic decoupled system by adding a third processor, to manage the interaction between the cache and main memory.

Superscalar processors can decouple the memory access and computation instructions to some degree: if the instruction window is large enough and a separate functional unit can execute memory accesses from the window, then the processor can hoist memory instructions (particularly loads) to execute earlier and hide some of the effect of long-latency cache misses.

2.8 Dependence-based architectures

Palacharla [42] identifies the issue window logic and data bypass paths within an out-of-order superscalar as being mostly likely to form the critical path as processors scale for future designs. A set of superscalar architectures are proposed – ‘dependence-based architectures’ – that reduce complexity compared to a standard superscalar while maintaining similar per-clock performance by breaking up these critical logic blocks.

Dependence-based architectures reduce complexity by partitioning the issue window and execution resources into multiple clusters. Since each of these clusters has a smaller instruction window and fewer data bypasses than a single large processor would, the complexity is reduced, enabling higher clock frequencies. In order to maintain IPC, instructions are steered to clusters in such a way as to minimise inter-cluster communication while still utilising all execution resources.

One specific dependence-based architecture outlined is a FIFO-based architecture. The issue window in an out-of-order superscalar is replaced by a set of FIFOs into which instructions are placed after decoding. Instructions issue from the FIFO heads, greatly reducing the complexity of the dependency-checking logic – only the instructions at the FIFO heads need to be checked, rather than all instructions as in a standard architecture. In order to maintain performance, dependent instructions are preferentially steered to the same FIFO, so as to minimise stalls at the FIFO heads. IPC is somewhat reduced from a standard model (about 5%), but the reduction in complexity should allow higher clock speeds or reduce power consumption.

2.9 Instruction-level distributed processing

Instruction-level distributed processing (ILDP) [49; 50] describes a set of techniques used to combat the problem that, as T_{clk} decreases and relative die sizes increase, wire delays on-chip become more significant to the point where it will no longer be possible to send a signal across a whole chip in one clock cycle. These techniques build upon the dependence-based architectures outlined in the previous section.

ILDP architectures break an instruction trace into multiple dependent execution streams, attempting to minimise the number of data values that must be communicated between streams. These streams can then be issued to separate functional units on-chip, with little communication required between these functional units. This distributed approach, which minimises global communication, reduces power consumption. The Alpha 21264 [18] uses a technique along these lines – there are two ALU 'clusters', each with its own copy of the register file. There is a one-cycle penalty for accessing a value from another stream.

Kim and Smith [30] present a specific ILDP architecture based upon the FIFO-based architecture from [42], using an accumulator-based instruction set to allow instruction dependencies to be explicitly highlighted within the code, simplifying the task of the instruction issue hardware. The architecture specifies that each functional unit executes in-order, but that they run independently of each other, so that performance approaches that of an out-of-order processor, without the same level of complexity. This architecture is described in much greater depth in Chapter 3.

2.10 Multiprocessing

Another way to exploit TLP is via *multiprocessing*. If one processor can execute two programs in a certain time, two processors should theoretically be able to manage it in half that time (as long as the programs are approximately the same length). In practice communication and synchronization delays tend to reduce the scaling to below linear. Multiprocessing produces very good results when a workload can be represented as many independent threads. When there are fewer threads than processors, when threads need to communicate frequently or a few threads are much longer than the rest combined, the performance can drop.

Multiprocessor architectures can be identified by how the processors are connected and the ways in which they access memory.

2.10.1 Symmetric multiprocessing

Symmetric multiprocessing (SMP) architectures link together multiple processors on a single memory bus. SMP architectures are relatively simple to design and provide good performance for low numbers of processors, but do not scale well as memory contention becomes a serious problem for large numbers of processors. On-chip caches can cause issues for SMP systems – additional logic must be provided in order to make sure that the contents of the caches represent a consistent view of memory and that processors cannot simultaneously write to the same location in (cached) memory.

2.10.2 Non-uniform memory architectures

Non-uniform memory architectures (NUMA) provide separate memory local to each processor – processors can access non-local memory, but at a greater latency than local memory. When the programs running on the processors access mostly only local memory, there can be a great performance gain due to the reduced memory contention, but the programming model is more difficult to work with and the requirements on the operating system are greater. NUMA scales better than SMP – with suitable software and an appropriate communication interconnect, a NUMA can be built with hundreds of processor nodes. A specific type of NUMA is Cache-coherent NUMA (ccNUMA), where cached copies of memory regions are kept consistent. With these systems there can be a significant performance penalty if multiple processors attempt to access the same region of memory concurrently.

While having a distributed memory organisation, a NUMA still generally has a single address space – nodes access non-local memory in the same fashion as local memory, but at a greater performance penalty. *Clustered computing* architectures contain multiple processing nodes (possibly themselves multiprocessors) which communicate over a more conventional communication network. Clustered architectures can be scaled to very large numbers of nodes (the IBM Blue Gene/L [14] supercomputer has 65,536 processing nodes), but typically require very specialised programming to fully utilize their capacity.

Some applications, typically scientific computing on very large datasets, provide enough parallelism to be able to run effectively on a large-scale multiprocessing system – or even require one. However, general-purpose computing often does not benefit beyond 2-4 CPUs. Some applications can only run as a single-threaded program, and will therefore not benefit from a multiprocessor architecture. Multiprocessing provides good performance gains where there are several threads running independently. In the case where threads communicate a lot, memory latencies become a performance bottleneck. Depending on the latencies involved in transferring data between CPUs, and the penalties involved in concurrent access to areas of memory, multiprocessors (particularly NUMA systems) generally cannot support high levels of coupling between threads.

2.10.3 Multicore processors

Multicore processors are a particular implementation of SMP where multiple processor cores are integrated onto a single chip. As feature sizes become smaller, placing multiple processors on a single die becomes feasible, and ameliorates the problem of global wire delays in modern processors becoming a limiting factor in performance, as most communication on-chip will be local to a processor core. As feature sizes continue to shrink, more processors can be added, making a cost-effective performance scaling option. The latest generations of desktop processors (e.g. Intel's Core Duo [16] and the AMD's Athlon X2) have moved to providing two processor cores on one die, with the cores sharing the L2 cache. Intel are planning to release a quad-core processor in the near future. Since the processors are connected at a higher level in the memory hierarchy (between L1 and L2 caches) than SMP or NUMA systems, they perform better with tightly-coupled threaded workloads.

Multiprocessing is not really a microarchitectural technique, as it is independent of the precise types of processor – almost any general-purpose CPU can be used in a multiprocessor system. Multiprocessing is more of a 'superarchitectural' technique. As such, it is orthogonal to the techniques being considered in this thesis; I concentrate on the efficient extraction of ILP from a single execution thread. There is some discussion of how this research could be applied to SMT- or SMP-style applications in Chapter 6.

2.11 Instruction encoding

In order for a processor to be able to execute a program, the instruction sequence must be represented in a binary form that can be loaded into memory. The way in which instructions are represented by binary data is known as the *instruction encoding*. There are two main categories of instruction encoding, relating to two different approaches to instructions sets, known as CISC and RISC.

2.11.1 CISC

Complex Instruction Set Computers, or CISCs, are generally characterised by large, feature-rich instruction sets with complicated variable-length encodings. What is now known as CISC design arose from previous generations of computer architectures – in very early architectures using core memory, memory access was slow and so implementing functionality as instructions rather than subroutines gave a performance improvement [43]. The trend continued even with the development of faster memories; programs written in assembly language benefited from a richer instruction set, as the effort required on the part of the programmer was reduced, and additional instructions could be marketed as extra processor features. The classic example of

a 'high-level' complex CISC instruction is the VAX 'polynomial evaluate' instruction, which computes the result of an arbitrary size polynomial given a parameter and a table of coefficients in memory.

The development of cache memories and high-level language compilers nullified many of the performance advantages present in CISCs – compilers often could not take advantage of complicated instructions, and instead would synthesise sequences of simpler instructions. In some cases these sequences executed faster than the complicated instructions they were replacing. Analyses of code execution patterns showed that on the IBM 370 architecture, approximately 15% of the instruction set accounted for over 90% of executed instructions [43]. The logic required to decode and execute these complex instructions, and the extra state required in order to service an interrupt partway through the execution of a long instruction greatly increased the complexity of designs, increasing their power consumption and production cost.

2.11.2 RISC

The Reduced Instruction Set Computer (RISC) was developed in response to many of the above problems. By removing complex and rarely-used execution logic, and focusing on an instruction set well-matched to the needs of the compiler, the cycle time could be greatly reduced. The logic area saved in making the execution logic simpler allowed other features such as pipelining and on-chip caches to be included – particularly useful with the limited transistor budgets of early single-chip VLSI designs.

The instruction encodings chosen for RISCs typically encode all instructions as 32-bit words, since this enabled a single instruction word to be fetched each cycle from instruction memory. Fixed bitfields within the instruction word define the opcode and operands. These features simplify the instruction decoding logic, but decrease code density as all instructions use four bytes regardless of the number of bits actually needed to encode them – whereas the CISC x86 instruction set can encode instructions in as little as one byte and averages under four.

Modern x86 architectures tend to be implemented as a RISC processor with a code translation engine which breaks complex instructions up into smaller RISC-like operations (referred to as μ-ops on the Pentium 4 [23]).

2.12 Embedded system design

Embedded processor designs can require comparable levels of performance to standard processors, but there are often strict power or space budgets. Techniques to improve performance on desktop systems can be entirely inappropriate for embedded systems. Until recently the fo-

cus in desktop development has been on performance over almost all else, while the embedded sector has looked into power-efficient performance: the benchmarking unit for embedded applications is not MIPS – it is MIPS/Watt. In the last year or two, desktop processors have been moving toward more power-efficient designs – the new Intel Core line of processors is based on the mobile variant of the Pentium 4, with the peak power consumption of the Core 2 Duo (a dual-core chip) being half that of previous high-performance Pentium 4 designs.

The main constraints present with embedded processor design are the needs for low cost, low power consumption and often the ability to provide real-time performance guarantees.

2.12.1 Low cost

Embedded systems often have very tight cost constraints. Even some of the most costly embedded systems, such as modern high-specification PDAs or mobile phones, have a similar retail price for the whole system as a high-performance desktop CPU on its own.

The recurrent cost of producing a die is equal to the cost of manufacturing a wafer, divided by the number of functional dice that the wafer contains. Using a Poisson defect distribution the die yield of a process depends on the area of each die, A, and the defect density of the process, D, as given in the following equation [64]:

$$Yield \propto e^{-AD} \tag{2.2}$$

For mature processes and reasonably small dice, the product AD is very small and the yield scales approximately linearly with die area – this is generally true when the number of dice per wafer is much larger than the expected number of defects per wafer. For larger dice, where the number of dice per wafer is comparable to the expected number of defects per wafer, the yield falls exponentially as die size increases. For low-cost systems, it is important to keep the die area within the high-yield linear zone.

Packaging costs are another per-unit cost on top of die costs. For a physically small system, it is advantageous to have as few chips as possible, reducing the size of the overall circuit board. Modern embedded systems often use a methodology known as *System-on-chip* (SoC), where the processor is combined not only with caches but also peripherals and sometimes memory all on a single die. This can give a cost advantage as well as a space advantage – as long as the die size is relatively small, combining two designs onto a single die does not increase the die costs significantly (and reduces mask costs), but does reduce the number of packages required and eliminates the need for package pins and circuit board area that would have connected those devices together. The additional requirement to fit various peripheral circuits on-chip in addition to the processor, while keeping the die size cost-effective, further constrains the maximum circuit complexity and size of embedded processors.

On top of per-unit production costs, setup costs for chip production are very high, running into millions of dollars. In order to make the chip cost viable for a low-cost embedded design the volume of chips manufactured must be in the order of millions. For designs with lower target volumes than this, a custom SoC is not possible. For these applications ‘platform chips’ are used – a SoC is designed targeting a range of products, being sufficiently general to be used in various applications. For example a television set-top-box platform chip might consist of a general-purpose CPU, a video encoding/decoding hardware block, an IDE interface for DVR applications, USB and/or IEEE-1394 controllers for I/O and possibly Ethernet or other networking hardware. A system designer could take one of these chips and build their design around it. Buying a generic platform chip and tailoring software to requirements greatly reduces costs.

Another aspect contributing to the cost of a system is the amount of memory present. Embedded systems often have very small amounts of RAM and FLASH. Better code density allows more code to fit in the same space, or the same code to fit in less space. This factor has slowed the uptake of RISC designs for embedded systems, as RISC architectures tend to have poor code density. CISC-based embedded CPUs such as the Motorola 68000 family became popular as a result of this.

To tackle this problem, various RISC architectures have been modified to include an embedded-friendly denser instruction set – for example MIPS16 [32] and ARM Thumb [3] add execution modes where the instructions are encoded in 16 bits rather than 32. The 16-bit execution modes usually only expose a subset of the instructions and registers present in the 32-bit mode. These encodings reduce the size of each instruction by half, but increase the number of instructions necessary, so the practical savings are listed as being around 40% for MIPS16 or 25% for ARM (the 32-bit ARM instruction set is denser than that of MIPS, so the gains with ARM Thumb are smaller). Since the insructions are less powerful and more are required, overall performance can drop using these reduced encodings, requiring a system designer to balance between code size and speed – although a number of embedded systems use 16-bit wide flash memory, giving an additional performance penalty for fetching a wider instruction. The ARM Thumb-2 ISA [44] allows 16-bit and 32-bit instructions to be mixed without changing the processor operating mode in order to provide good code density while maintaining high performance. IBM have introduced CodePack [13] for the embedded PowerPC architecure, which compresses blocks of instructions.

2.12.2 Low power consumption

A typical battery in a mobile phone or PDA may have a capacity of 1500mAh at 3V, which translates to a total energy of 16200J. A modern desktop processor can draw over 100W of power – at 100W, that 1500mAh battery will last less than three minutes! While this is an extreme example, it highlights the vast difference in the power constraints on embedded systems compared to desktop systems. Even ‘low-power’ desktop processors such as Intel’s latest Core

2 consume in the order of 65W. A processor suitable for a battery-powered PDA or phone must draw less than 1W in order to last for hours on a 1500mAh battery.

Heat dissipation is another issue strongly related to power consumption. The CPU drawing 100W will radiate most of that power as heat and thus will require a large heatsink. Without this heatsink it will overheat and fail – tests have shown that with the heatsink removed, some AMD Athlon processors can reach a surface temperature in excess of 300°C [41]. Embedded systems may have tight physical size constraints, preventing the use of large heatsinks, and may also have a limit to the amount of heat that can be generated, due to problems such as limited airflow preventing heat dissipation (e.g. handheld devices being stored in users' pockets).

Power consumption is generally represented as the sum of *dynamic* power consumption and *static* power consumption. Dynamic power consumption represents the power consumed during switching activity – the power expended charging or discharging output loads as gates turn off or on. Static power consumption is any power consumed by the circuit independent of activity, through leakage currents etc. These quantities are given by the following equations [64]:

$$P = P_{dynamic} + P_{static} \tag{2.3}$$

$$P_{dynamic} \propto f_{CLK} \cdot V_{DD}^2 \tag{2.4}$$

$$P_{static} = I_{static} \cdot V_{DD} \tag{2.5}$$

Power consumption can be reduced by lowering the supply voltage V_{DD} – dynamic power scales quadratically with supply voltage. Decreasing supply voltage increases the circuit propagation delay, however, as given in Equation 2.6 [31] (α is a constant with a value of approximately 1.3 with current technology). In order to maintain performance with a reduced V_{DD} the threshold voltage V_{TH} must also be lowered to reduce the propagation delay.

$$T_{pd} \propto \frac{V_{DD}}{(V_{DD} - V_{TH})^{\alpha}} \tag{2.6}$$

There are problems associated with lowering the threshold voltage; reducing V_{TH} also reduces the noise margin and increases leakage current. Subthreshold leakage current increases exponentially as the threshold voltage is reduced, as shown in Equation 2.7, where n is a process-dependent constant and V_{θ} is the thermal voltage, which is approximately 26mV at room temperature.

$$I_{leak} \propto e^{\frac{-V_{TH}}{n \cdot V_{\theta}}} \tag{2.7}$$

An embedded design must make a trade-off between scaling V_{DD} to reduce power consumption but harm performance, and scaling V_{TH} to regain performance but increase static power consumption.

Driving off-chip signals uses a large amount of power, as the I/O pads and attached circuit traces have a very high capacitance. SoC designs where memory and peripherals are integrated on-chip reduce power consumption as on-chip interconnects have a much lower capacitive load than off-chip connects over circuit boards. High-activity signals such as memory buses can provide a large power saving if moved on-chip.

Inactive circuitry contributes not only to static power through leakage, but also to dynamic power, as clock edges will still cause transistors to switch, consuming power. This is a problem as such circuitry is consuming power without contributing to performance. The power consumed by inactive circuitry can be reduced by techniques such as *clock gating* and *power gating*. Clock gating reduces the dynamic power consumed by gating clock signals so that they do not propagate into inactive blocks. Power gating removes all power to a circuit block, reducing both the dynamic and static consumption. The disadvantages of these methods are that they tend to increase the time required to reactivate an inactive block, depending on the granularity of the gating. Circuits using fine-grain clock gating can be powered up very quickly, and if the gating is used effectively across a chip, the power savings can be substantial.

2.12.3 Real-time performance

General-purpose processor design tends to focus on increasing average performance – for example a deep pipeline may allow a high clock speed, but will increase the branch misprediction penalty. The average performance will increase, but the worst case performance can actually drop. In many embedded systems it is necessary to provide real-time response, thus these systems must meet tight worst-case performance bounds. Modern architectural techniques such as branch prediction and memory caching can create a large disparity between typical and worst-case performance, and often operate in a non-deterministic fashion, making it very hard to predict what the actual performance will be.

2.13 Future developments

Modern high-performance processors achieve high clock speeds by increasing pipeline depth and reducing the amount of logic in each pipeline stage; the Pentium 4 [23] uses a 20-stage pipeline. Hartstein and Puzak [19] and Hrishikesh et al. [24] investigate the limits of such developments – and show that current architectures are approaching those limits. Some believe that these limits have already been reached; the AMD Athlon and the latest IBM PowerPC

both provide comparable performance to the Pentium 4 with shallower pipelines and lower clock rates. Intel's latest generation of processors provide higher performance and lower power consumption at a lower clock speed.

Burger [6] provides an introduction to the problem that, over the course of the next decade, the relative increase in on-chip wire delays will require new architectures in order to achieve performance gains. Agarwal et al. [1] study this in more detail and it is shown that, if conventional architectures continue to be used, the current annual performance improvement of 55% will drop to the region of 12% on future technologies.

3 ILDP in an Embedded Context

This chapter describes the Instruction-Level Distributed Processing (ILDP) microarchitecture as outlined by Kim and Smith [30] and evaluates it in the context of an embedded system. The complexity of the main architectural blocks of the design are compared qualitatively against a simple scalar architecture and a typical out-of-order superscalar architecture.

3.1 Details of ILDP

Consider the following line of C code, part of the Tiny Encryption Algorithm:

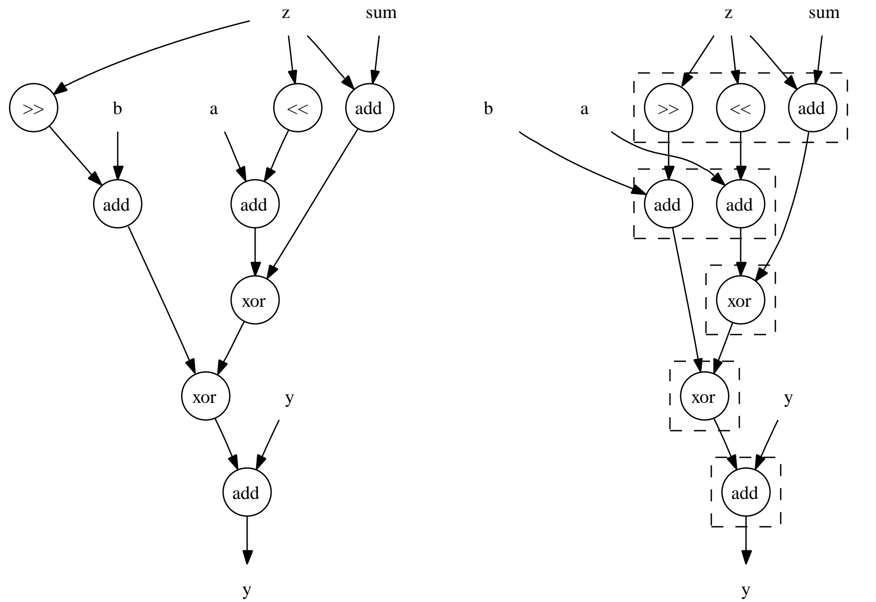
```
y += ((z << 4) + a) ^ (z + sum) ^ ((z >> 5) + b);
```

Ignoring any necessary loads/stores, this line of code can be represented by instructions with the dependency graph shown in Figure 3.1(a). In order for a standard out-of-order superscalar processor to execute these instructions in as few cycles as possible, it needs to identify which instructions are independent of each other and thus can be issued simultaneously. Figure 3.1(b) redraws the graph, with instructions that can be issued together grouped. It can be seen that each group contains no internal arcs, as there are no interdependencies between the instructions within a group. Two instructions within the same group can always be issued together.

Another way of representing the dependency graph is to group the instructions into streams where each instruction depends on its predecessor in the stream (except for the first instruction) so that the instructions in a group form a chain. This attempts to minimise the number of arcs between groups – an example is shown in Figure 3.1(c). Here the instructions are grouped vertically rather than horizontally. Two instructions that can be executed at the same time must come from different groups.

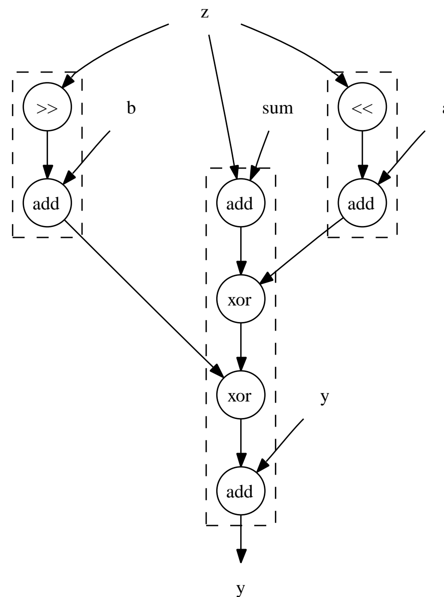
The main novel feature of the ILDP architecture is the hierarchical register file – as well as a set of general purpose registers, there are several accumulator registers. ALU operations must target and source an accumulator, and cannot reference more than one accumulator. To use values residing in two different accumulators, one of those values must be moved into a GPR. This contrasts to the standard RISC approach in which a three-operand model is used:

Figure 3.1 Ways of grouping instructions by their dependencies



(a) Dependency graph

(b) Graph with independent instructions grouped



(c) Graph grouped into instruction streams

instructions (other than loads and stores) read one or two registers (or immediate constants) and then write to a register, using a single (mostly) orthogonal register set. With the ILDP system, typical instructions are two-operand – the destination (an accumulator) must also appear as a source.

While a standard superscalar processor needs complex issue logic in order to identify independent instructions which are ready for execution, ILDP instructions which form a dependency stream will use the same accumulator. In effect, the vertical groupings of Figure 3.1(c) are explicitly highlighted to the processor by the compiler's register allocation; each group will be assigned a different accumulator. The processor can readily identify such dependent streams and issue instructions from different streams separately. This simplifies instruction issue, effectively turning the dependency analysis in a reorder buffer on its head.

3.2 Instruction set

Kim and Smith's ISA specifies eight accumulators and 64 GPRs, which for this research have been specified as 32 bits wide. Each instruction can use a single GPR as either a source or a destination, but not both. This simplifies the register rename logic, as shown in Section 3.6.1.

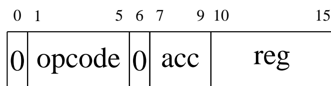
A stream of dependent instructions is referred to as a *strand*. All the instructions within a strand will access the same accumulator register. Accumulators can be *live* or *dead* – a live accumulator is one that is currently being used by an instruction strand and thus contains useful data. Initially all accumulators are marked as being dead. When an instruction accesses a dead accumulator, or writes to a live accumulator without reading from it, this starts a new instruction strand. When an instruction with the '.c' (for 'close') suffix to its opcode is executed, the accumulator is marked as dead – thus streams of dependent instructions can be explicitly highlighted as such within the instruction stream, simplifying the processor's dependency analysis. This is covered in more detail in the next section.

Instructions are encoded into 16-bit words referred to as *parcels*. Each instruction can be encoded as one or two parcels – giving a variable-length instruction encoding, with 16 or 32 bits per instruction. The instruction encoding formats are summarised in Figure 3.2. Kim and Smith [30] does not fully describe the instruction set or encoding details, so a full instruction set and binary encoding have been created based on what is specified.

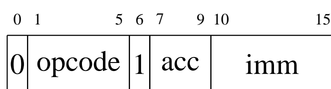
The instruction set is summarised in Table 3.1. The assembler syntax specified is quite verbose, redundantly specifying source and destination operands that are the same – this was chosen for clarity in assembly listings. In the 'semantics' column, A_i refers to the ith accumulator, R_j refers to the jth GPR, *imm* is an immediate constant which, depending on the instruction format, can be 6, 16 or 22 bits wide. The various lengths of immediate allow instructions using small (common) values to be encoded in a single parcel, reducing code size. For the branch

instructions, *label* refers to a program location to jump to and is encoded in the immediate field as a PC-relative displacement – a branch to the instruction directly following it would have an immediate field containing a displacement of zero.

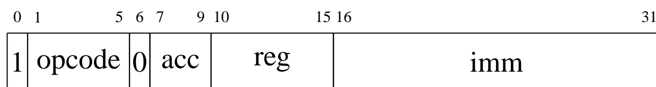
Figure 3.2 Instruction encodings



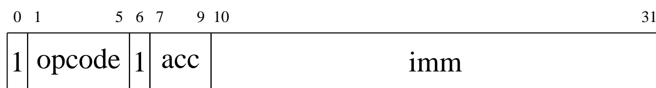
Short register format



Short immediate format



Long register/immediate format



Long immediate format

3.3 Programming model

While not a part of the hardware architecture, the Application Binary Interface (ABI) chosen for this work is described here. The ABI model is based loosely on that of the ARM APCS [9] and shares features in common with many different architectures.

3.3.1 Register assignments

The architecture specifies 64 GPRs, of which r63 is used to hold the return address after a call instruction. Table 3.2 shows the assigned roles for each of the registers in the ILDP ABI. The first sixteen registers are used to pass arguments to functions and to return result values.

The stack pointer points to the bottom of the stack, while the frame pointer points to the top of the current function's stack frame; this is described further in the next section.

Syntax	Semantics
Dyadic ALU instructions; *op* is one of add, sub, rsb (reverse subtract), and, ior (logical or), xor, mul	
<op> a<i> <= a<i>, r<j>	$A_i := A_i \ \mathit{op} \ R_j$
<op> a<i> <= a<i>, #<imm>	$A_i := A_i \ \mathit{op} \ \mathit{imm}$
<op> a<i> <= r<j>, #<imm>	$A_i := R_j \ \mathit{op} \ \mathit{imm}$
<op>.c r<j> <= a<i>, #<imm>	$R_j := A_i \ \mathit{op} \ \mathit{imm}$
Shift instructions; *op* is one of sll (shift left logical), srl (shift right logical), sra (shift right arithmetic), rol (rotate left)	
<op> a<i> <= a<i>, r<j>	$A_i := A_i \ \mathit{op} \ R_j$
<op> a<i> <= a<i>, #<imm>	$A_i := A_i \ \mathit{op} \ \mathit{imm}$
Monadic ALU instructions; *op* is one of abs, inv	
<op> a<i> <= r<j>	$A_i := \mathit{op} \ R_j$
<op> a<i> <= a<i>	$A_i := \mathit{op} \ A_i$
Comparison instructions; *cond* is one of eq, ne, le, lt, ge, gt (all signed)	
c<cond> a<i> <= a<i>, r<j>	$A_i := A_i \ \mathit{cond} \ R_j$
c<cond> a<i> <= a<i>, #<imm>	$A_i := A_i \ \mathit{cond} \ \mathit{imm}$
Move instructions	
mov a<i> <= r<j>	$A_i := R_j$
mov a<i> <= #<imm>	$A_i := \mathit{imm}$
mova a<i> <= @<label>	$A_i := \mathit{label}$
mov r<j> <= a<i>	$R_j := A_i$
mov.c r<j> <= a<i>	$R_j := A_i$
Load/store instructions	
ldr a<i> <= [a<i>+#imm]	$A_i := \mathit{MEM}[A_i + \mathit{imm}]$
ldr a<i> <= [r<j>+#imm]	$A_i := \mathit{MEM}[R_j + \mathit{imm}]$
str [a<i>+#imm] <= r<j>	$\mathit{MEM}[A_i + \mathit{imm}] := R_j$
str [r<j>+#imm] <= a<i>	$\mathit{MEM}[R_j + \mathit{imm}] := A_i$
Branch instructions	
b<cond> @<label> ? a<i>, r<j>	$\mathit{PC} := \mathit{label}$ iff $A_i \ \mathit{cond} \ R_j$
b<cond> @<label> ? a<i>, #0	$\mathit{PC} := \mathit{label}$ iff $A_i \ \mathit{cond} \ 0$
br @label	$\mathit{PC} := \mathit{label}$
Jump/call instructions	
jmp a<i>	$\mathit{PC} := A_i$
jmp r<j>	$\mathit{PC} := R_j$
call a<i>	$R_{63} := \mathit{next_PC}; \mathit{PC} := A_i$
call r<j>	$R_{63} := \mathit{next_PC}; \mathit{PC} := R_j$
Other instructions	
nop	Does nothing
syscall	System call

Table 3.1: The ILDP instruction set

Register(s)	Usage
a0–a7	Scratch registers, not preserved over function call
r0–r15	Argument and result registers
r16–r31	Callee-saved; preserved over function call
r32–r59	Scratch registers, not preserved over function call
r60	Frame pointer
r61	Assembler temporary
r62	Stack pointer
r63	Link register

Table 3.2: Register assignments for the ILDP ABI

The accumulators and most of the GPRs are not preserved over a function call – it is the caller's responsibility to save these registers where necessary. The GPRs r16–r31, the frame pointer and the stack pointer are preserved – in this case a function must ensure that it saves and restores the values of these registers if it needs to change them while executing.

3.3.2 Memory organisation

A fairly standard program memory model is used. Program code is located at the lowest portion of virtual memory, with the dynamically allocated heap starting above that. There is a full descending stack – the stack grows downwards, and the stack pointer points at the location holding the most recent stack value.

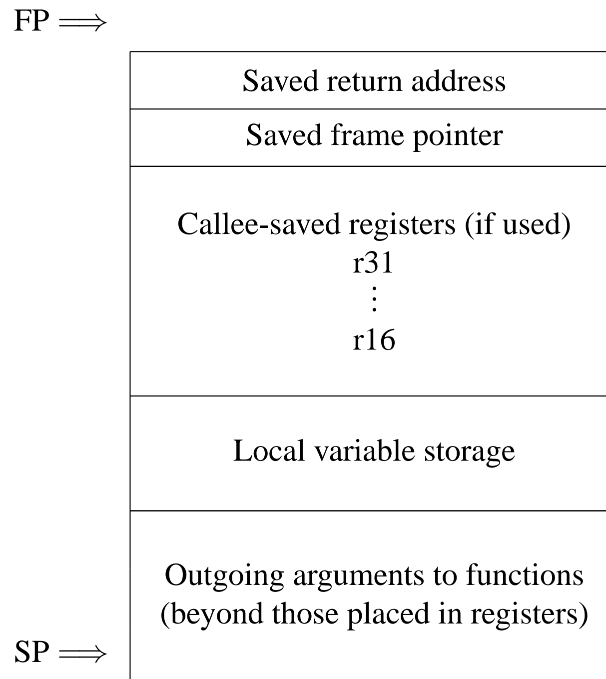
3.3.3 Function call specification

A function is invoked using the `call` instruction. On entry to the function, r63 contains the return address, r62 points to the stack and function arguments are provided in r0–r15 (if these registers are not sufficient to hold all of the arguments, additional arguments are passed on the stack).

The function sets up a stack frame to hold its return address, any register values that need saving and any local storage to the function that may be required – the frame layout is depicted in Figure 3.3. The frame pointer is set up to point at the top of the frame (usually equal to the initial value of the stack pointer, except for 'vararg' functions) and the stack pointer is decremented to point to the new stack bottom.

On return from a function, any results are placed in r0–r15, saved registers are restored and the stack frame is popped, then a `jmp` to the return address is executed.

Figure 3.3 Function stack frame layout



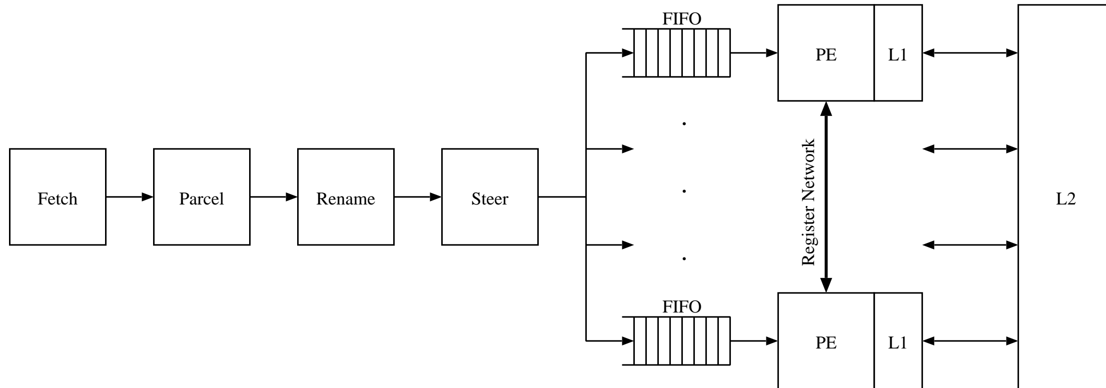
3.4 Microarchitecture

The ILDP architecture is outlined in Figure 3.4. The front-end fetch and issue pipeline is four instructions wide, and is split into four stages:

1) Instruction fetch
2) Decode (termed ‘parcel’ as the fetched parcels have to be reassembled into complete instructions)
3) General-purpose register rename
4) Instruction issue/steering (also known as accumulator renaming)

The front-end pipeline issues instructions to a set of eight processing elements (PEs) where they are executed, and then a standard in-order commit mechanism is used to retire instructions. In this implementation, all parts of the design run at the same speed from a single clock, but due to the decoupling between the issue pipeline and the PEs, these could potentially be clocked at different speeds.

Figure 3.4 The ILDP architecture, as outlined by Kim and Smith



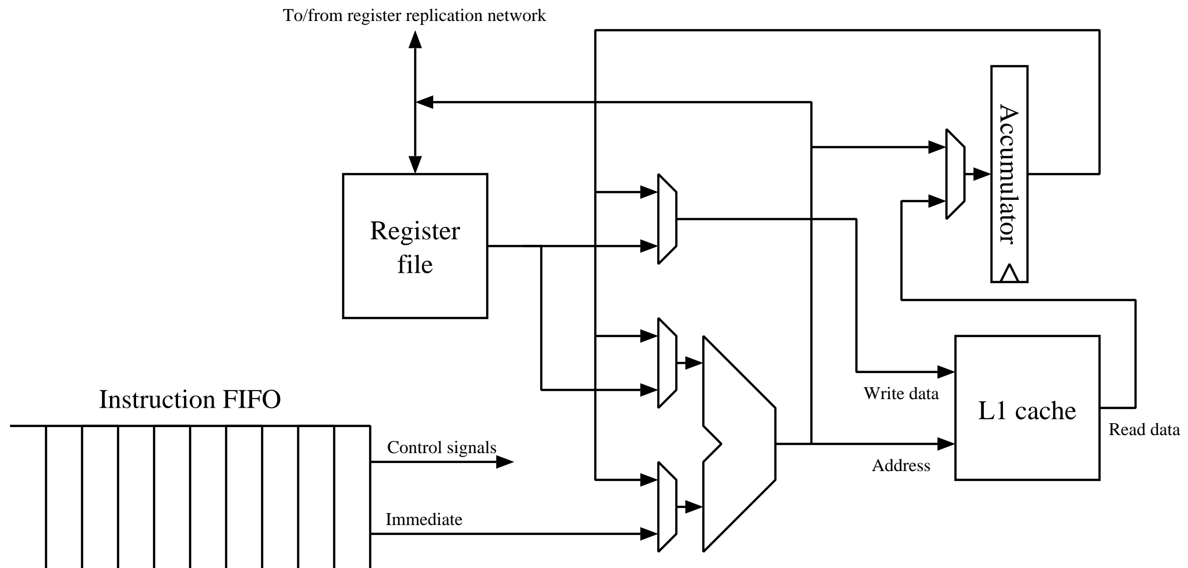
The instruction fetch pipeline stage loads a 128-bit word from the instruction cache and places it into a fetch buffer. The fetch buffer is 192 bits wide and holds any undecoded parcels from the previous cycle (the parcel logic can consume between 64 and 128 bits in a cycle) as well as the newly fetched instruction word. The fetch logic aligns the new data correctly when placing it in the buffer, so that newly fetched parcels appear in sequence immediately after older ones.

The parcel pipeline stage analyses the fetched data in 16-bit blocks and reassembles them into four instructions – this procedure is made relatively simple as the first bit of a parcel indicates whether the instruction is encoded in 16 or 32 bits. Since the instructions are variable length, often not all of a 128-bit word will be consumed at once – the parcel logic can store 'leftover' parcels back into the fetch buffer and assemble them in the following cycle along with a newly fetched word. The parcel stage also performs decoding of the instruction opcode.

In the register rename stage, any of the 64 GPRs used by the instructions are renamed onto a physical set of 128 registers, using rename logic as in a standard superscalar – although the fact that most instructions use one GPR (or none) rather than two or three as in a standard processor allows the rename logic to be simpler and reduces the number of ports required on the rename map. The pipeline can stall in the rename stage if the register rename map fills – the issue pipeline will then wait for instructions to commit and free up physical registers for renaming.

The steering pipeline stage allocates instruction strands to processing elements by renaming the eight ISA accumulator registers onto the eight physical accumulators within the processing elements. When an instruction starting a new strand is processed by the steering logic, it is allocated to a free processing element (if there is more than one, the one with the emptiest issue FIFO is chosen). Further instructions for a strand are issued to whichever PE is currently executing that strand. After issuing the final instruction of a strand (flagged as such using the '.c' suffix to the opcode), the PE is marked as being available for allocation once more.

Figure 3.5 Structure of an ILDP processing element



The structure of a processing element is shown in Figure 3.5. Each PE contains an ALU and a copy of the register file and memory access logic, as well as a single accumulator and an instruction FIFO. Each processing element executes instructions in order from its FIFO. As the instructions executed by each PE form dependent chains, the ALU in the PE is not pipelined, simplifying its implementation somewhat. Register values and the addresses of store instructions are replicated between PEs using a communication network. Instructions that use a register value can only issue if that value is present in the local register file – i.e. if the value has been received over the register communication network. The exact form of the communication network is not specified as part of the architecture, as ILDP is designed to be tolerant of a high communication latency, allowing a range of implementations. For this research the network is modelled as a bus with a 2-cycle latency (mirroring that used in [30]).

Load instructions can only issue if there are no earlier store instructions to the same address in flight. This is enforced by having a copy of the store queue in each of the PEs – store instructions are allocated store queue entries in the rename pipeline stage. When a store instruction executes, it communicates the memory address and stored value over another communication network similar to that for register values.

Each PE requires its own memory access port. This can be implemented by having a separate data cache for each PE, with a coherency network to keep the cache contents equal (the approach described in [30]) or by having multiported caches shared between PEs, or by multiplexing cache ports between PEs. Multiplexing cache ports between PEs (with associated arbitration logic) is likely to reduce performance compared to the other options, as the effective bandwidth to memory is reduced.

3.5 Complexity

This architecture should provide several benefits over an out-of-order architecture in terms of complexity. Since each processing element only communicates values through the register file, far fewer forwarding paths are required – forwarding paths are only needed within a PE, not between them as in a standard architecture. The inter-PE communication network can be implemented in various ways depending on performance and area requirements, whereas standard forwarding paths scale quadratically with issue width and linearly with pipeline depth.

As instructions are issued in-order within PEs, and issued to PEs depending on their accumulator, the issue logic will also be far simpler than a full superscalar processor, eliminating the need for a reorder buffer etc. Thus the complexity of the ILDP architecture should be similar to that of an in-order superscalar architecture, but should provide performance similar to an out-of-order system [30]. This should make it well suited to embedded applications, and is analysed in greater depth in this section.

Fully analysing the complexity of a processor design is a large and complicated task, generally requiring implementation of the whole design at the circuit level. Most microprocessors are full custom designs, so to accurately assess complexity of architectural modifications a full custom design would be necessary, which is beyond the scope of many research projects. A simpler approach is a more qualitative analysis, breaking down the design and looking at the complex structures that are likely to affect the critical path and how they scale with feature size and architectural parameters. In such an analysis, Palacharla [42] investigated the following components of modern CPU designs:

- Register rename logic
- Issue window dependency checking logic
- Instruction issue logic
- Register file
- Data bypass paths and logic

Palacharla identifies the issue window logic and bypass paths in a superscalar processor as being most likely to affect performance in future architectures due to not scaling well with feature size. The ILDP architecture eliminates the complex reorder buffer of a superscalar architecture in favour of multiple small simple issue FIFOs.

3.6 Complexity analysis

The following sections analyse the components listed above, looking at the blocks contributing to their complexity, and how architectural parameters affect complexity. The analysis is performed for typical examples of:

- The ILDP design
- A superscalar architecture
- A simple scalar architecture

3.6.1 Register rename logic

Register rename logic is composed of several blocks:

- The *map table* which holds the current mappings between logical and physical register numbers, generally implemented as a RAM indexed on logical register number. There is also a table holding previous register mappings, used when the processor has to roll back execution after an exception.
- A free register list, holding the tags of all physical registers available for allocation.
- Logic to allocate new registers from the free register list to instructions that write to logical registers.
- Logic to detect interdependencies between instructions in the block currently being renamed and identify those that should read physical registers allocated in the current cycle instead of those specified in the map table.
- A multiplexer to select the correct physical register numbers for each instruction and pass the results along the pipeline.

The size of the map table scales with the number of logical registers in the ISA. Its complexity also depends on the issue width of the processor and the number of operands per instruction in the ISA.

A typical superscalar processor of issue width n with instructions that have r input operands and w output operands will require $n \times r$ read ports and $n \times w$ write ports to the map table. A 4-way issue processor with a typical 3-operand instruction set will have a map table with 8 read ports and 4 write ports.

By contrast, the ILDP architecture can either read or write one register per instruction, so for 4-way issue just 4 read/write ports would be required, eliminating eight decoders and sets of word lines compared to a 4-way issue processor. This reduction in logic and wires should reduce power consumption and also make the block smaller, reducing the overall delay. A separate set of renaming logic is used in the steering stage to rename the accumulator registers – this also requires only 4 read/write ports, and will only contain eight entries.

A scalar CPU does not use register renaming, and thus can entirely eliminate rename logic. The ILDP architecture will necessarily be more complex than a scalar processor in this regard.

3.6.2 Issue window dependency logic

In a standard superscalar, the instruction window holds all in-flight instructions. Instructions can only be issued once all their input operands are available; the dependency checking logic is responsible for flagging instructions ready as these operands become available. Each cycle the physical register numbers of the data that has been computed are checked against the waiting instructions via an associative lookup – any instructions that match are marked as such. Once all operands are available, an instruction can be issued.

The instruction window in a superscalar processor is implemented as an associative memory. Each memory row can hold one instruction, with the associative section containing the data tags for the instruction's operands and bits indicating when operands are available. An n-way issue processor requires n associative ports for writing newly available data tags, and each row then needs n comparators per operand to check for matches. The larger the issue width, the larger each row will be, and thus the length of the bitlines in the memory will grow. Overall the delay of this block is quadratic in both the issue width and window size, and at smaller feature sizes the timing tends to become dominated by wire delays and thus scales poorly with modern processes.

The ILDP architecture eliminates the instruction window; instead instructions are placed into FIFOs and issue from the FIFO heads. Dependency checking only needs to be done on the instructions at the heads of each FIFO, and as each instruction only uses a single register operand, there needs to be only a single comparator per processing element, rather than 8 per row of the instruction window as in a 4-way superscalar (a 64-instruction window would then require 512 comparators and associated logic). This should make the instruction window logic much smaller in the ILDP design – the total size of the instruction FIFOs will be similar to that of the superscalar instruction window, but the dependency checking logic is greatly reduced.

A scalar processor has no instruction window logic (and also no instruction FIFOs).

3.6.3 Instruction issue logic

A standard superscalar must identify instructions available for execution and allocate them to functional units for execution, with the oldest such instructions being executed first. The dependency checking logic above will mark instructions as being available, and then the issue logic is generally constructed using an *arbiter tree*. This has an $O(\log n)$ time complexity, and a space complexity of $O(n \log n)$, with n being the instruction window size.

The ILDP architecture, in contrast, issues instructions into the set of FIFOs for the processing elements. The logic for this is basically a second set of register renaming logic, this time renaming the accumulator registers onto the set of processing elements. As the set of FIFOs will generally be small (e.g. eight FIFOs in [30]), this logic block (termed the steering logic) should be small and fast compared to an arbiter tree for a large instruction window – it will certainly be faster than the logic for renaming general-purpose registers, due to the smaller map size, so should not form part of the critical path.

Within a PE, the issue logic simply executes the instruction at the head of the FIFO if it either does not depend on a register value, or depends on a register value that is available (see 3.6.4).

In a scalar processor, the issue logic generally consists of a scoreboard to track which register values are currently within the register file and which are in flight, along with logic to choose which forwarding paths (if any) to use and interlock logic to halt the pipeline, inserting a bubble, when there are data hazards requiring a stall.

3.6.4 Register file

A standard processor architecture will require one read port to the register file for each input register operand of each executing instruction and one write port for each output operand. This is basically the same as the register rename map – for a processor with a 4-way execute bandwidth, executing 3-operand instructions, 8 read ports and 4 write ports will be required to the register file. In order to reduce the register file complexity, several superscalar architectures make multiple copies of the register file, with fewer read ports per file, e.g. an 8 read port, 4 write port register file could be split into two copies with 4 read ports and 4 write ports each. Writes are replicated to each copy of the register file, while reads are distributed across the copies. This increases the total die area usage, but reduces the delay of the register file logic.

Since each PE in the ILDP architecture has a separate register file, each individual file needs fewer ports than a superscalar processor with a single file – this is similar to a superscalar processor with a split register file. Each instruction can either read or write a single register, and register values can be received over the register communication network. As such, the ILDP processor requires only one read/write port and enough write-only ports to handle the bandwidth

of the register network on each register file (the read/write port could possibly be replaced by a read-only port, as register writes from the PE must go through the register network anyway, and could be written using one of the dedicated write ports).

This compares favourably with a scalar RISC architecture, which will generally have two read ports and one write port. The ILDP register files should therefore be small (for the number of registers they contain) and fast, but require replication across each PE, increasing total silicon area for the design.

3.6.5 Bypass paths

A standard superscalar processor with issue width w and n pipeline stages requiring bypass (any stages from the first execution stage before register writeback) will need $2nw^2$ bypass paths and $2n$ multiplexers to select the appropriate path for use. For modern systems with deep pipelines, complex logic and large issue widths this leads to having many long wires with associated delays. As feature sizes drop, these delays scale badly and can easily become the limiting factor on performance.

The ILDP architecture separates the bypass paths into local paths within a PE and the global register value communication network between PEs. Each PE requires only one or two bypasses, which should be fairly short (and thus fast). The ILDP architecture is designed to be tolerant of delays in the the register communication network – [30] shows only a small performance degradation when the delay is increased from zero to two cycles. This tolerance allows various different implementations to be used for this interconnect, depending on area, power and cycle time constraints.

A scalar processor will also have $2nw^2$ bypass paths, but with a w value of 1. For a typical five-stage pipeline, n is 2, giving four total bypass paths.

3.6.6 Data caches

Data caches are generally hard to scale for high issue widths – adding additional ports to a first-level cache tends to increase the delay significantly. Modern designs tend either to use a single-cycle L1 data cache with one read and one write port or a multi-cycle pipelined cache with two read ports and one write port. Superscalar designs often avoid the need to implement additional read ports by using *non-blocking* caches – a single-ported non-blocking cache does not need to stall when a read access causes a miss; later instructions can continue to use the read port while the miss is being serviced.

The ILDP architecture specifies that each PE has its own memory access port. This either requires each PE to have its own coherent copy of the L1 cache, or that multiple PEs share a multi-ported L1 cache. Individual caches per PE will be the highest-performance option, but will require a large silicon area – caches use the majority of silicon area on modern designs, and replicating the L1 cache eight ways will either use up eight times as much chip area, or reduce the size of the cache by a factor of eight. Sharing a larger multiported cache between PEs is likely to increase the cache latency, but in a situation where the silicon area is limited, the increased size of the cache may improve the hit rate, mitigating this penalty slightly.

A scalar processor will generally have a single L1 cache with one read/write port, which does not need to support non-blocking accesses.

3.7 Power consumption

The ILDP architecture reduces complexity in a number of key areas compared to a typical superscalar processor – as such, it should have a correspondingly lower power consumption. However, the power consumption is still likely to be higher than the scalar RISC CPUs typically used in embedded systems. Given the higher performance, some degree of additional power consumption is acceptable; battery life is traded off against speed. One important factor will be the resulting power efficiency – whether the performance increase is greater than the added power consumption.

While complex structures are eliminated or scaled down, many of the simpler structures are replicated across PEs. This will particularly have an effect on the static power consumption, as some of these (such as the register file) are fairly large blocks. Depending on the usage patterns of the PEs, it may be possible to use clock and/or power gating to reduce the power consumption – if many of the PEs are idle for large amounts of time, clock gating will reduce their dynamic power consumption and power gating will reduce static power consumption. This would greatly increase the power efficiency, as only those blocks that are active – and thus directly contributing to performance – would consume power. As blocks that have been powered down can have fairly large start-up times, it might be necessary for performance to keep one PE as a 'hot spare' – idle, but still powered, to hide the latency of powering up an inactive PE. Such a spare could still utilize clock gating to reduce power consumption, as clock gated logic can be restarted much quicker than power gated logic.

One of the structures replicated across each processing element is the register file. This replication will increase power consumption, potentially quite significantly, as the register file is a large component and thus will have a correspondingly high static power usage. While PEs can theoretically be powered down when idle to reduce their power consumption, the data in the register files has to be kept coherent and up-to-date. As such the register files cannot be

powered down – they will continue to consume not only static power, but also dynamic power due to writes distributed over the register replication network.

3.8 Summary

This chapter has shown that while the ILDP architecture utilizes many superscalar processing techniques, it eliminates or scales down many of the higher-complexity structures of a superscalar processor, bringing its complexity closer to that of a typical embedded processor, at an increased speed. The performance of the design is analysed quantitatively in the next chapter.

4 Evaluating ILDP

This chapter evaluates the performance of the ILDP architecture compared to typical embedded processors and explores the effect various architectural parameters have on performance.

4.1 Performance vs other embedded architectures

The initial investigation was to evaluate the performance of the ILDP architecture outlined in Kim and Smith [30] under embedded workloads. Their work executes Alpha binaries using dynamic code translation in the style of the Transmeta Crusoe [8; 33]. For an embedded system, however, dynamic translation at the hardware level adds extra complexity, especially since binary compatibility is generally less of an issue for these applications – particularly given the growing popularity of bytecode languages such as Java and .NET, which require dynamic interpretation or compilation regardless.

4.1.1 Methodology

To evaluate the ILDP architecture presented in Chapter 3, a set of benchmarks were run using a simulation of the ILDP processor and compared to results of the same benchmarks run on other architectures. Since the possible clock speed of a processor depends heavily on the detailed circuit-level design, which is beyond the scope of these evaluations, the results are compared based on the total number of clock cycles used to execute the benchmark workloads. Based on the complexity analysis in Section 3.6, it is assumed that similar clock speeds could be attained by ILDP and the other tested architectures.

4.1.2 Compiler toolchain

In order to evaluate the architecture, it was necessary to be able to generate instruction code to execute; thus it was necessary to target a compiler and associated toolchain to the new ISA. The GNU Compiler Collection (GCC) was initially evaluated but developing a new backend seemed

to be an overly involved process so other options were investigated. The SUIF [2] project looked promising, providing an open modular compiler framework, along with Machine-SUIF [53], a framework for creating compiler backends to generate machine code.

Using the SUIF framework, I created a basic compiler for the ILDP architecture. There were some problems adapting the register allocator to handle the hierarchical register file and so a new pass was developed for the compiler which would annotate the code with hints to allow register allocation to take place. This was able to compile simple test programs into the ILDP ISA. The New Jersey Machine-Code Toolkit [45; 46] was used to generate object code – given a high-level description of the instruction encodings, the NJMCT will generate code to generate instructions in binary format. The GNU BFD library (part of the binutils package) was then used to generate ELF object files containing the compiled code.

In order to compile any benchmark suites, a C library was required – the uClibc library seemed like an appropriate choice, as it is designed to be smaller and more readily portable than the usual GNU libc. When trying to compile the uClibc library however I found that my compiler system did not function well for compiling anything larger than basic test programs due to problems with data layout and linking, and that the library code contained a number of GCC-specific language features which were tricky to work around. I also found the C front-end to SUIF to have a number of problems compiling modern C programs. Unfortunately this front-end is based on a commercial compiler and is only available as a closed-source binary module, and thus cannot be modified. As such, I investigated alternatives to the main compiler front-end that could alleviate these problems.

Ultimately, a toolchain based on the GNU compiler and binutils was developed (the documentation in [40] proved invaluable), which compiles directly to the ISA of the simulated architecture. The CGEN [11] tool was used to generate an assembler, disassembler and a basic functional simulator from a high-level description of the instruction set – similar to the NJMCT used earlier, but the description includes not only the instruction encoding, but also the assembler syntax and semantics – making it much simpler to generate a full toolchain with minimal additional effort. The newlib C library was used, as this proved to be even simpler to port to new architectures than the uClibc library, and also supports 'multilib' compilation which allows several variants of the ISA to be used and to have toolchains automatically built for all variants; this proved useful for the work in Chapter 5.

The strand allocation in the compiler was handled using the existing GCC register allocation mechanism – for each instruction in the ISA, a pattern is provided to the compiler specifying the allowable register types for each operand. The compiler contains an internal constraint solver which tries to allocate registers in such a way as to maximise performance – usually by minimising the number of move instructions that need to be inserted. In practice, the compiler proved to have a certain amount of difficulty handling some of the constraints peculiar to ILDP – in particular, it proved very difficult to prevent the compiler from trying to generate accumulator-to-accumulator move instructions. Also, the constraint that a store instruction could either store an accumulator value to a register address or a register value to an accumulator address was

impossible to specify. A number of fixups had to be created to work around these compiler issues.

Ultimately the code generated by the compiler was often not optimal for the architecture – in order to improve the generated code the register allocator would need to made aware of the concept of strands. Unfortunately there was not enough time within the scope of this research to invest in further compiler developments. The performance of ILDP versus other architectures (with more mature and better optimised compiler backends) is likely to suffer as a result of this.

4.1.3 Benchmarks

I evaluated a number of benchmark suites to see which would be most suitable. The SPECint [20] suite is most commonly used for evaluating processor architectures, but is targeted at high-end systems – the SPECint2000 suite uses a working set of 256MB, too large for an embedded system for which typically the total system memory will be in the order of 64MB. I felt that a benchmark suite targeted specifically at evaluating embedded systems would be more appropriate for my research.

The EEMBC suite is the industry standard for embedded benchmarking, but is only available after purchase of a closed licence with severe restrictions on the publishing of results. I felt that this would probably be inappropriate for my initial research, and that something more readily available would be more beneficial for academic work.

MiBench [17] is a freely available embedded benchmark suite based on open-source software packages. I decided to use this suite for evaluation as it was free and seemed fairly easy to set up and use.

4.1.4 Simulation

The SimpleScalar simulation environment [5] was investigated for performance evaluation, but proved to be heavily tied in to a standard superscalar processor model – while it could be used to simulate the other architectures, it was not suitable to simulate the ILDP design. Instead, a custom cycle-accurate software simulator was created to run the code for the ILDP model. Each circuit block within the design was implemented as a block of software, suitably parameterized so that architectural parameters could be changed and evaluated. An execution framework was created that would allow automated execution of the benchmark suite over a range of parameters, and would allow running benchmarks on other architectures for comparison purposes.

For the initial experiments the architectural parameters of the ILDP model were set as in [30], i.e. with a 4-way issue/retire width and 8 FIFOs/PEs. The register communication network was

modelled as a simple bus between the PEs, with a latency of two cycles to transmit the values between PEs (plus an additional cycle for the PEs to write the data into the register file). The instruction FIFOs were set to be of unlimited size, and the register communication network could transmit an unlimited number of values in a single cycle. This allowed the simulations to run assuming these parameters had been set to a suitable level to allow as close to optimum performance as possible. Sections 4.3.3 and 4.3.4 explore the effect these parameters have on performance.

4.1.5 Results

The performance of the baseline ILDP architecture was simulated versus a model of the ARM SA1100 and a PowerPC model, chosen to approximately represent the PPC405 core. For the SA1100 model, the Simplescalar-ARM [5] simulation system was used, and for the PowerPC simulations Dynamic Simplescalar was used [25] – while Simplescalar-ARM is supplied with a configuration for the SA1100, the configuration for the PowerPC model had to be created based on published specifications [26], and the performance of these models may be less accurate as a result. A model for the PPC440 core [27] was attempted, but it proved difficult to set the parameters for the superscalar execution and it was not possible to test the accuracy of the model against a real implementation, thus making it impossible to use for comparison. The measured performance values are based on the number of cycles taken to execute each benchmark – it is assumed that the achievable clock speeds for these architectures will be similar. The benchmarks were compiled with the same libraries and toolchain for all architectures (the newlib C library and GCC version 3.3.2). Due to limitations in the set of the system calls implemented in newlib, not all of the benchmark tests would link – the results presented are from the 20 tests which could be compiled. The same set of compiler flags were used for each architecture – most of the tests were compiled using '`-O3`', but due to compiler issues the *tiff* tests had to be compiled with '`-O2`' and the *jpeg* tests were compiled using '`-O`'.

The results of these simulations can be seen in Figure 4.1 – the results shown are normalised to the average execution time for each test. It can be seen that, on average, there is a 18.5% speedup over the ARM. The *patricia* benchmark shows the greatest improvement – about 172% – while the *adpcm (enc)* and *tiffdither* tests actually run approximately 10% slower than the ARM. Compared to the PPC405 model, ILDP performs on average 52.8% better, ranging from 23.3% on *adpcm (enc)* to 104% better on *tiff2rgba*.

These results show that the 4-way issue ILDP model performs better than the single-issue ARM, but not by a very large margin – this is somewhat disappointing, as [30] shows the performance of ILDP to be similar to that of a 4-way out-of-order architecture. Figure 4.2 shows that in these simulations the ILDP architecture executed more total instructions than the other architectures – in some cases more than twice as many. This is attributed in part to the need for additional move instructions in the strand-based architecture, in part to the density of three-operand instruction code over the two-operand ILDP instructions (in an extreme case, an ARM can execute an

Parameter	ILDP	ARM	PPC405
Issue width	4-way	1-way	1-way
Execution width	8-way	1-way	1-way
Branch prediction	16K entry, 12-bit global history gshare	All branches predicted not taken	2K entry bimodal
L1 I-cache	Perfect 1-cycle latency	16KB 32-way associative, 32B line size, 1-cycle latency, fifo replacement	Perfect 1-cycle latency
L1 D-cache	8KB 2-way associative, 64B line size, 1-cycle latency, random replacement	16KB 32-way associative, 32B line size, 1-cycle latency, fifo replacement	32KB 2-way associative, 32B line size, 1-cycle latency, LRU replacement
L2 cache	256KB 2-way associative, 128B line size, 8-cycle latency, random replacement	none	256KB 8-way associative, 64B line size, 8-cycle latency, LRU replacement
Memory	72-cycle latency	72-cycle latency	72-cycle latency

Table 4.1: Architectural parameters for simulation

Figure 4.1 Performance of ILDP compared to embedded processors

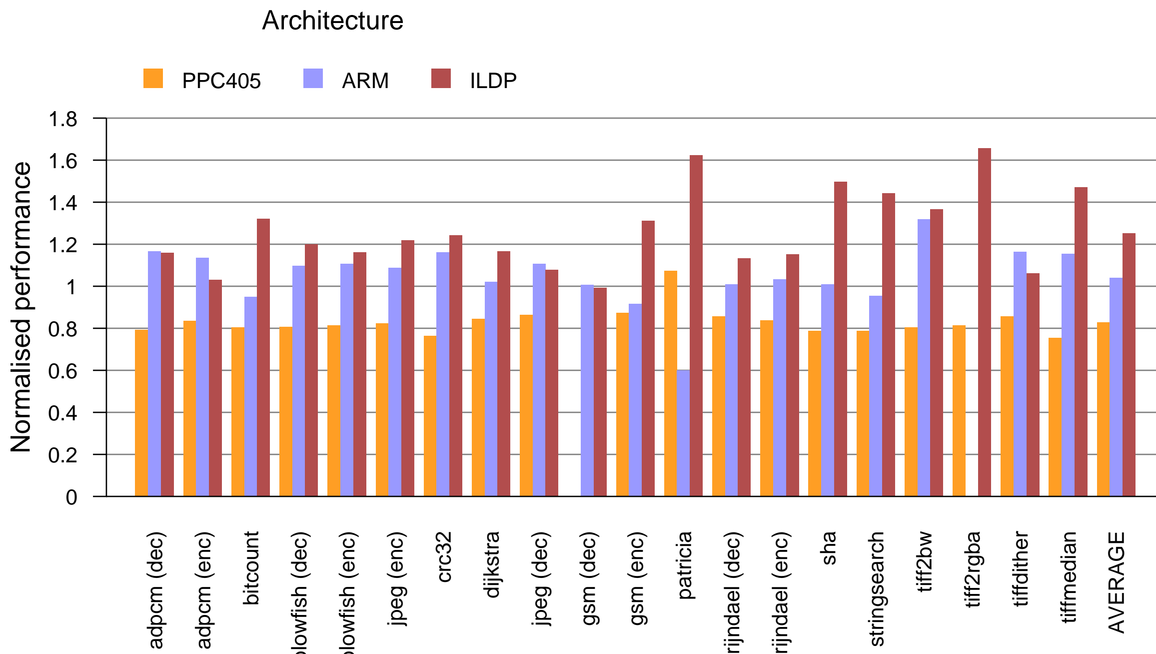


Figure 4.2 Number of instructions executed, compared to ARM

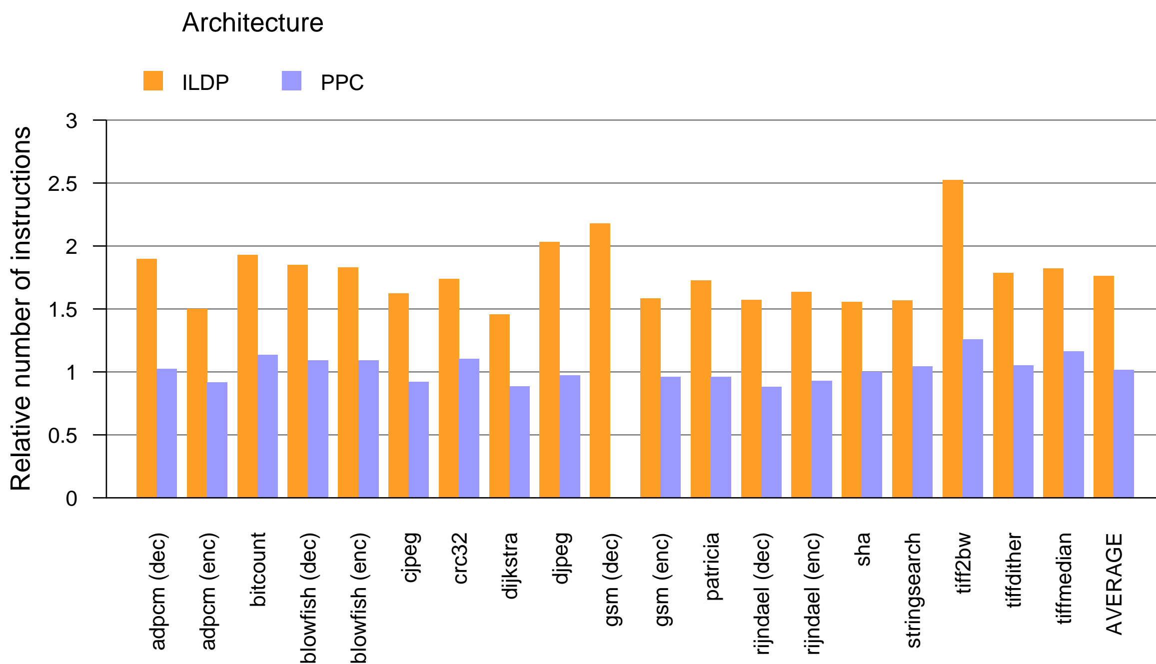
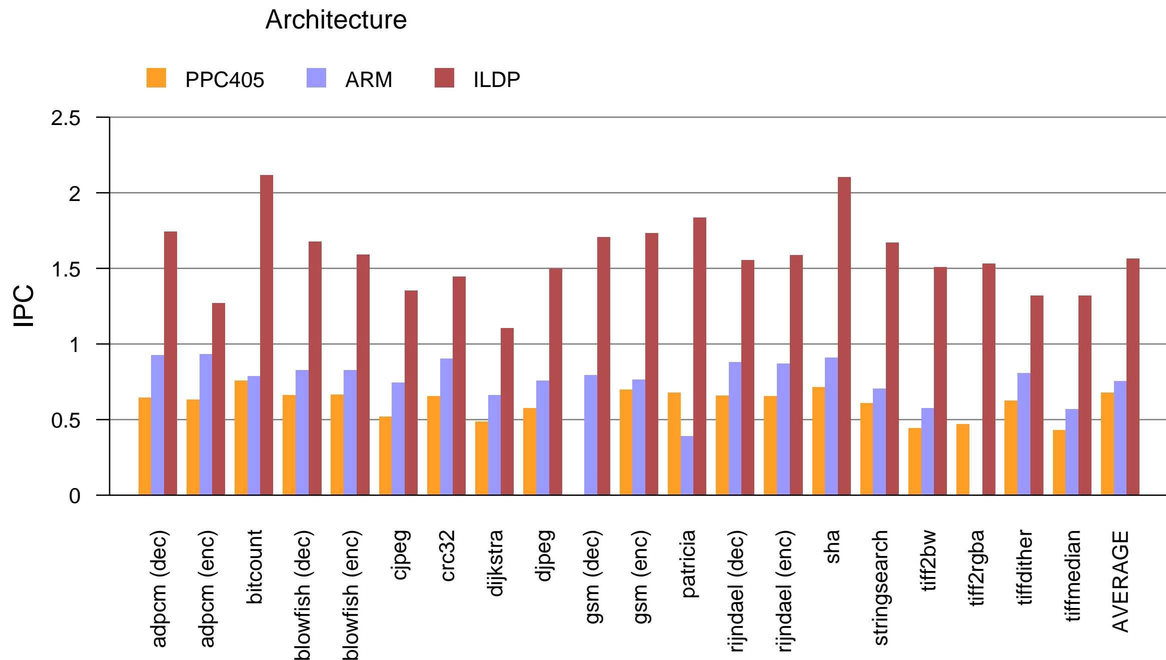


Figure 4.3 IPC of tested processors



arithmetic operation and a shift between arbitrary registers in a single instruction, while the ILDP can take up to four instructions for such an operation) and in part due to the compiler. The ILDP compiler is not optimised as well as the compilers for the other architectures – in the original simulations the *crc* benchmark ran approximately 10% slower than the ARM, but manual analysis of the code suggested two peephole optimizations which were introduced to the compiler, thus removing two instructions from the inner loop and increasing the overall performance. In particular, the compiler implementation proved to have problems with the lack of an accumulator-to-accumulator move instruction and the operand formats for the load/store instructions.

The IPC attained by the ILDP architecture is much greater than that of the other architectures (as shown in Figure 4.3) but performance is limited by the increased code size. An interesting comparison would be running the ARM tests on code compiled for the Thumb architecture, as the Thumb ISA has a number of restrictions similar to ILDP – shifts are performed as separate instructions and most ALU operations are represented as two-operand instructions. Unfortunately the SimpleScalar-ARM simulator cannot execute Thumb code, so this comparison was not possible.

4.1.6 Conclusion

Overall, while the performance of the tested ILDP architecture is only moderately greater than the embedded processors used for comparison, the level of IPC achieved is significantly higher – assuming that the compiler could be further optimised in order to reduce the total number of instructions executed, the ILDP architecture should show much better performance.

4.2 Resource utilization

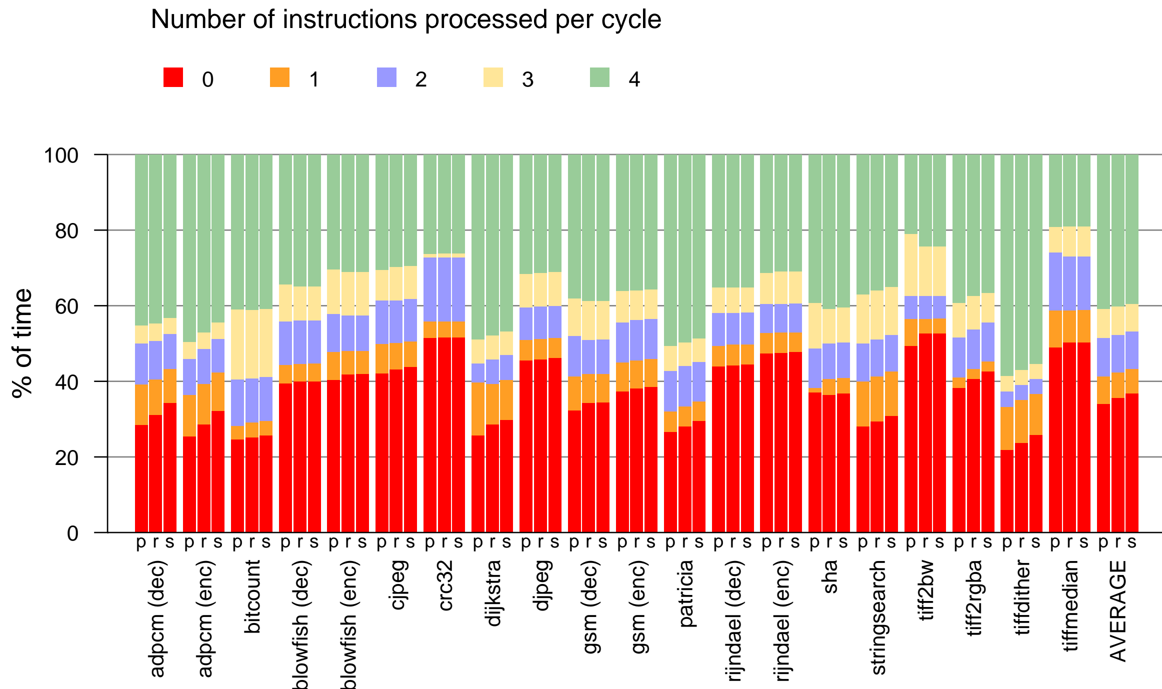
As manufacturing moves to smaller feature sizes, static power consumption is becoming more and more of an issue. Idle circuitry wastes large amounts of power – circuits should either be utilized or powered down in order to get maximum power efficiency. Investigations were thus made into how fully hardware resources are being utilized. If a hardware block is replicated in order to support a certain issue width or execution width, but simulation shows the utilization to be low, then possibly the width for that block is too high and the level of replication could be reduced without significant effect on performance.

4.2.1 Decode / Issue logic

The complexity of many components in the processor scale linearly or quadratically with the issue width of the processor. If the limiting factors in performance come from other aspects of the system – e.g. the execution width, delays from branch mispredictions or cache misses, or code containing too little inherent parallelism – then having too high an issue width can mean the issue pipeline is largely idle, wasting power and can reduce overall performance if the logic is on the critical path and the added complexity increases the cycle time.

Figure 4.4 shows the number of instructions processed by each stage of the issue pipeline proportionally in time. When running the benchmark suite on the 4-way issue model, it can be seen that the parcel (decode), rename and steer (issue) pipeline stages spend on average around 35% of their time completely idle. This is attributed to the level of parallelism present in the benchmarks – there is simply not enough ILP present in the code to sustain 4-way issue, thus the issue logic runs ahead of the execution logic and will stall waiting for instructions to be consumed. Analysing the IPC achieved on these benchmarks corroborates this – as shown in Figure 4.3, the average IPC is only 1.56, and only the *bitcount* and *sha* tests achieve more than 2 IPC. These results indicate that the issue pipeline is only being utilized at about half capacity and that a 2-way issue pipeline might be sufficient to maintain performance on these benchmarks. If the issue width is too high, then the issue pipeline is using more logic than necessary, which will increase static power consumption. Section 4.3.1 explores the effects of issue width on pipeline utilization.

Figure 4.4 Number of instructions processed by the parcel, rename and steer pipeline stages

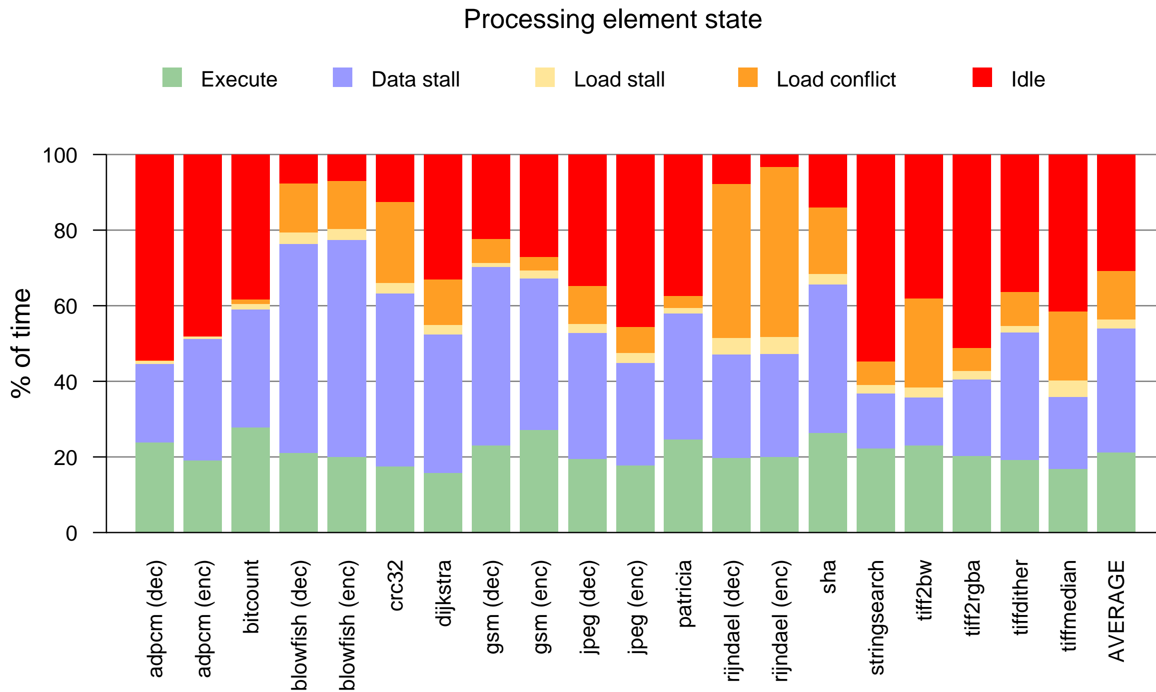


4.2.2 Processing elements

As above, if the number of processing elements is too high compared to the other architectural parameters, then they will be idle most of the time, wasting power and silicon area. A plot of the activity of the individual processing elements (Figure 4.5) shows that only about 21% of the time is spent executing instructions – most of the time is spent idle, waiting on GPR values from other PEs (data stalls), or waiting on loads that are blocked by earlier stores (load conflicts).

The fact that the PEs spend much less than half their time executing implies that complexity savings could be made by having fewer PEs (e.g. four instead of eight), and associating two instruction FIFOs and accumulators with each one. The PE would then execute from both queues in a round-robin fashion – for most of the time one of the queues would be empty or stalled, so there would be only a minor performance impact. Since each processing element is independent, those PEs with empty FIFOs could be powered down which would allow high peak execution rates while saving power. This option is explored in Section 5.1.

Figure 4.5 Processing element activity



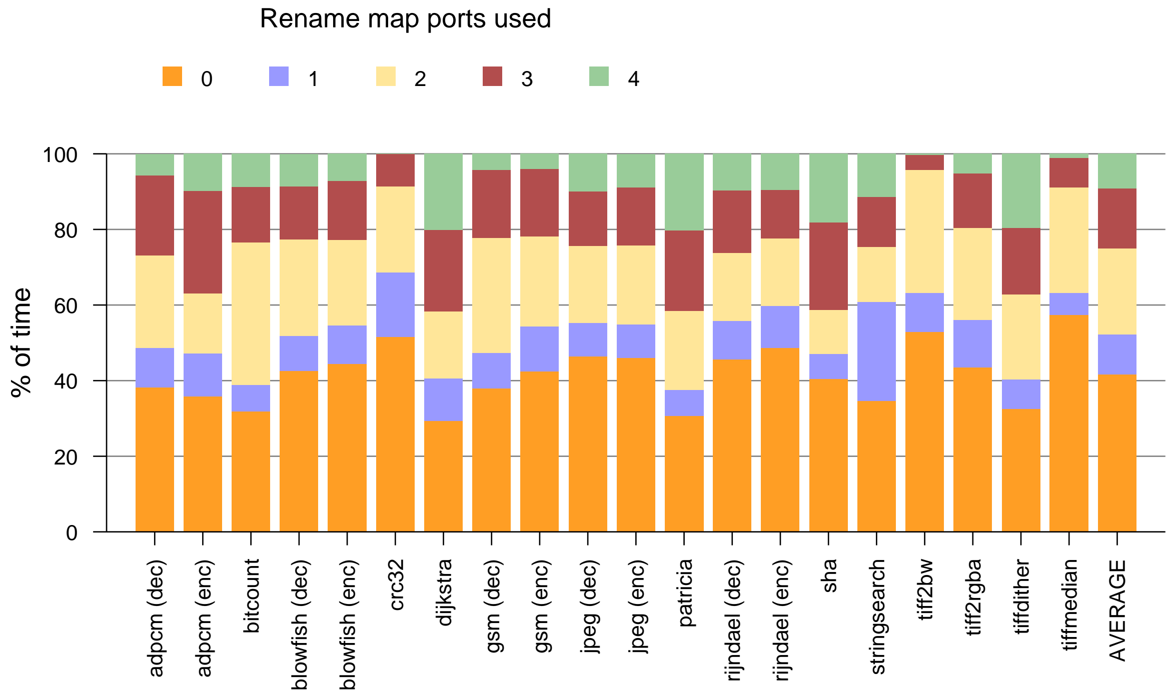
4.2.3 Register rename logic

The rename logic in the ILDP design is simpler than that of a standard superscalar, as the instructions reference up to one register operand rather than three. The ILDP rename logic provides one read/write port to the rename map for each instruction in the rename stage of the pipeline. Depending on the proportion of instructions accessing register values in the instruction stream, it is possible that the rename logic could be further simplified by providing fewer ports, and stalling rename of instructions when the available bandwidth is exceeded. Figure 4.6 shows the distribution of rename map bandwidth usage over time.

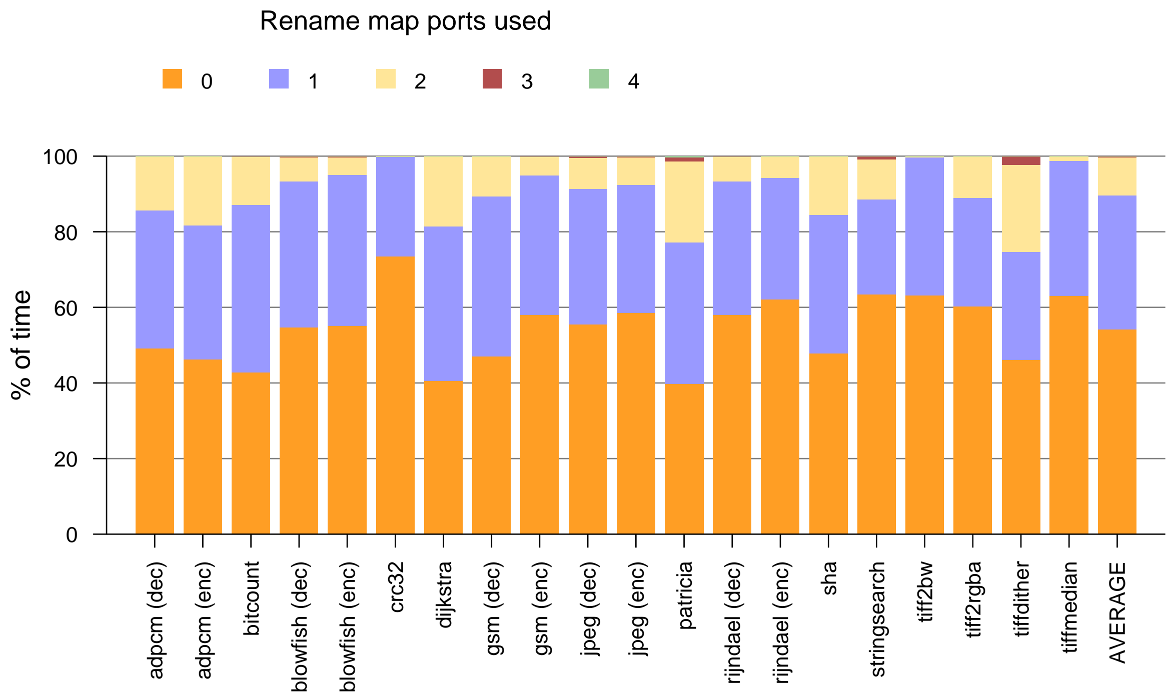
It can be seen that in a large proportion of cycles, no registers are renamed – this is mostly due to the fact that the issue pipeline is idle for a significant fraction of the time, as shown in Section 4.2.1.

These results show that even when the rename logic is active, it is still not utilized to full capacity for much of the time. On average, the rename stage uses all four rename ports in only 9.1% of cycles. Three ports are used 15.9% of the time, two ports for 22.6% and one port for 10.6%. These results imply that if the complexity of the rename logic were a problem even with the simplifications inherent to the ILDP architecture, the logic could be scaled back to only rename three registers per cycle with likely little effect on performance. Scaling to a

Figure 4.6 Distribution of rename map bandwidth usage over time



(a) Overall rename map port usage



(b) Rename map port write usage

rename width of two per cycle could have a more significant effect. This is evaluated further in Section 4.3.2.

Figure 4.6(b) shows the usage of the rename map for writes only. This clearly shows that far fewer ports are used for writing to the rename logic – for nearly 90% of the time one or fewer writes are performed to the rename map, and more than two writes are performed in less than 0.4% of cycles. If the complexity of implementing all read/write ports is significantly greater than implementing some ports as read-only, then reducing the write bandwidth to two is likely to cause a negligible performance reduction, and reducing to a single read/write port may also be a reasonable performance trade-off.

4.3 Parameter space exploration

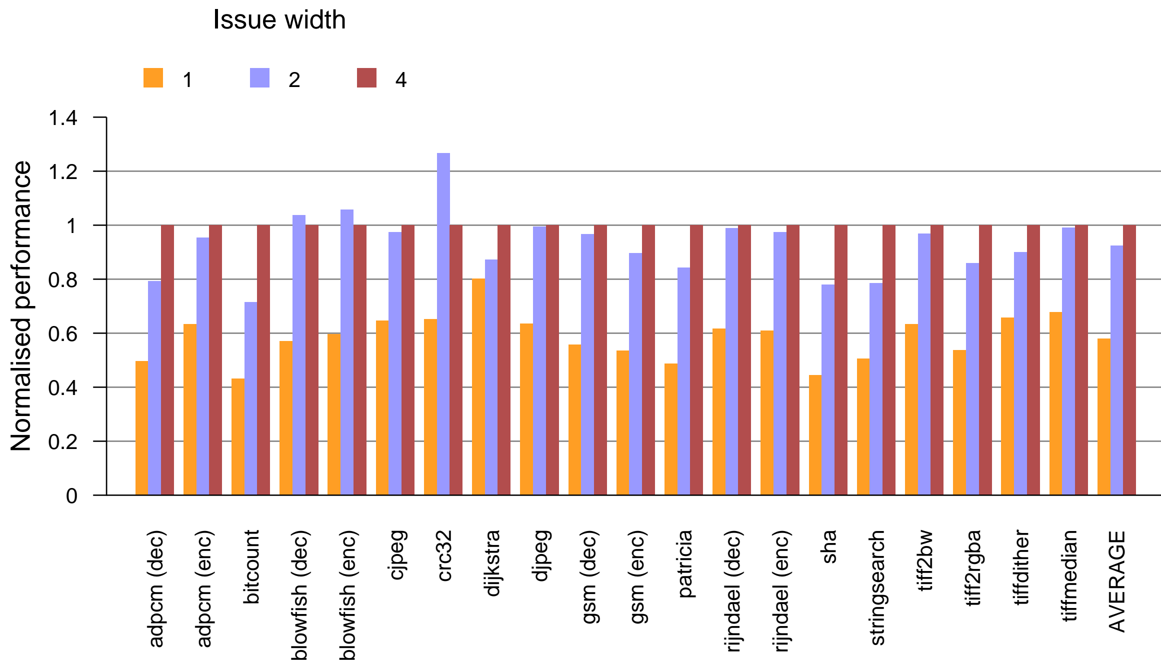
As well as evaluating the performance of the ILDP architecture using the parameters specified in [30], simulations were run exploring various different architectural parameters to see if a particular combination would be better suited to the constraints of embedded design – reducing the potential complexity and/or power consumption without significantly impacting upon performance.

4.3.1 Issue width

Experiments were conducted varying the issue bandwidth – as well as the original 4-way issue model, 2-way issue and single issue were evaluated. Figure 4.7 shows the impact on performance due to scaling the issue width. Going from 4-way issue to 2-way gives on average an 7.6% drop in performance, although there is a significant variation across the tests – the *bitcount* and *sha* tests both lose more than 20% performance. This is unsurprising as both these tests were able to average more than two instructions per clock on the 4-way architecture – as the tests with the highest level of parallelism achieved, they are likely to lose out most from the issue width reduction.

Another result of note from this graph is that the *crc32* benchmark actually performs over 25% better when the issue width is width is set to 2-way issue. This result is unexpected and is reflected to a smaller degree in the *blowfish* tests, which show a slight performance improvement going from 4-way to 2-way issue. Analysis of the simulator statistics does not show any obvious explanation for these results, and so it is assumed that for some of these tests, a larger issue width allowing deeper speculative execution can actually limit performance when execution paths are mis-speculated – instructions are incorrectly executed and can, for example, cause cache conflicts and pollute the cache with irrelevant data. At a lower issue width, the processor would execute less deeply down speculative paths, and so be affected to a lesser degree. The

Figure 4.7 Performance of ILDP at various issue widths, normalised to 4-way case



crc32 benchmark would then represent a pathological example of this. If this data point is taken to be an outlier, the performance drop from going to 2-way issue is about 9.2%. Moving to single-issue has a much larger impact, with performance dropping on average by nearly 42%. This reflects the results from Section 4.2.1, where a 4-way pipeline was running at around two-thirds capacity.

Figure 4.8(b) shows the pipeline utilization with 2-way issue: significantly less time (21%) is spent idle, compared with 4-way issue as described earlier (Figure 4.4 is reprinted as Figure 4.8(a) for reference). Having less logic which is utilized more in order to provide comparable performance will improve the power efficiency of the issue pipeline.

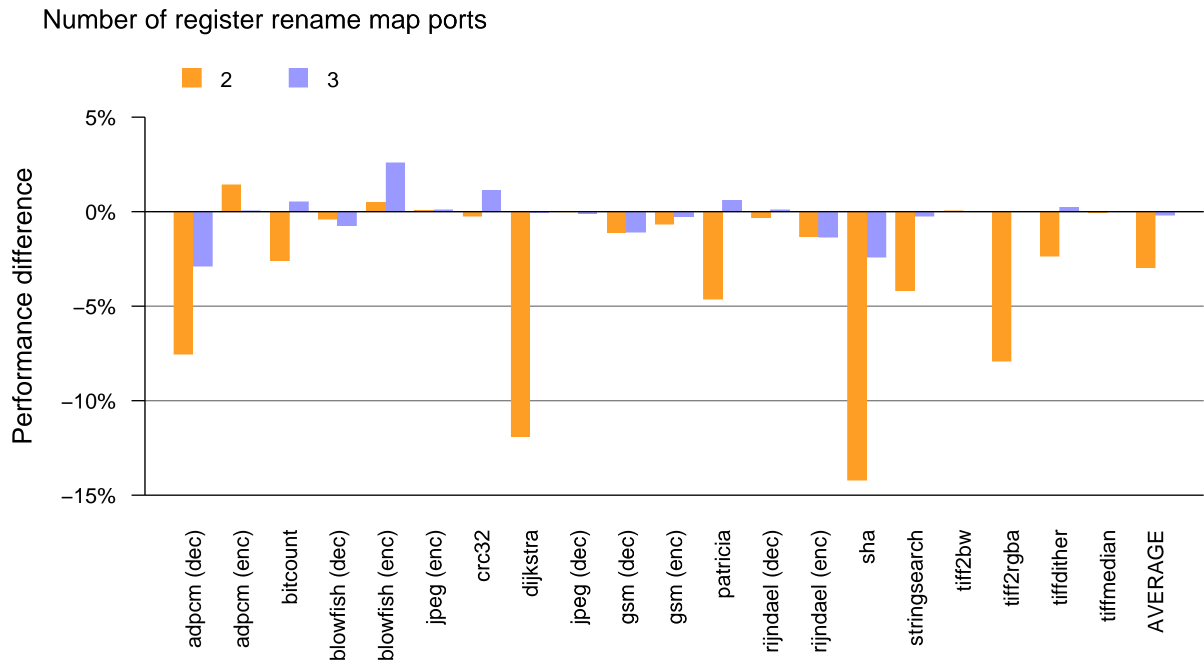
It is possible that the reduction in complexity of the issue logic in a 2-way issue design would allow for higher clock speeds as well as reducing power consumption, mitigating the IPC penalty and possibly allowing the 2-way architecture to perform better overall. Even if this is not the case, the 2-way issue architecture is reasonably close to the 4-way design in terms of performance, and will be more power-efficient. Depending on system design constraints, either could be suitable for a particular design. The single-issue design does not perform very well – since the ISA and execution logic are designed with superscalar execution in mind, a single instruction-per-clock issue pipeline will prove to be a severe bottleneck. For the rest of this chapter, both the 2-way and 4-way issue designs are evaluated.

Figure 4.8 Number of instructions in the parcel, rename and steer pipeline stages

(a) 4-way issue

(b) 2-way issue

Figure 4.9 Effect on performance of removing register rename map ports



4.3.2 Rename bandwidth

If the added complexity of the rename logic due to issue width turns out to be greater than that of the other pipeline stages, it may be beneficial to reduce the width of the rename stage while leaving the rest of the issue pipeline at a 4-way width. Section 4.2.3 shows that for much of the time the full number of ports on the register rename map are not used. This section evaluates the effect on performance of reducing the number of ports on the map, thus limiting the peak rename bandwidth. If instructions enter the rename stage requiring more ports than are available, the later instructions will have to be stalled. Experiments were conducted varying the number of rename map ports.

Figure 4.9 shows the effect on performance of reducing the number of ports implemented in the rename map. When the number of ports is reduced to three, very little variation in performance can be observed. The worst affected test is *adpcm (dec)* which performs slightly under 3% slower. On average, the three-port case achieves 99.8% of the performance of the baseline 4-port case. With only two rename map ports, a larger performance reduction is observed – while many tests perform to within 1% of the baseline model, some tests are more than 10% slower (the *sha* test performs 14% worse). On average, with only two rename map ports, 97% of the 4-port performance is still achieved.

Analysing the number of instructions processed per cycle in the issue pipeline shows that the 3-port model (Figure 4.10(a)) is, on average, still able to process four instructions in a cycle

Figure 4.10 Issue pipeline statistics when varying number of rename map ports

(a) 3 rename map ports

(b) 2 rename map ports

approximately 35% of the time (compared to 40% in the base model), and the 2-port model (Figure 4.10(b)) processed four instructions in 18% of cycles, and three or more instructions in 37% of cycles on average. This indicates that there are are many instructions that utilize no GPRs (e.g. monadic instructions or those using an immediate operand) in the instruction stream, and that they are distributed fairly evenly within the code.

The reduced dependence of ILDP instructions upon the general-purpose register file means that implementing fewer ports into the register rename map does not cause a significant variation in performance in most tests. If the cycle time in a particular implementation were limited by the complexity of the register rename map and the additional logic required to arbitrate for fewer ports were not a problem, reducing the number of ports into the map would be a performance-effective approach to reducing complexity.

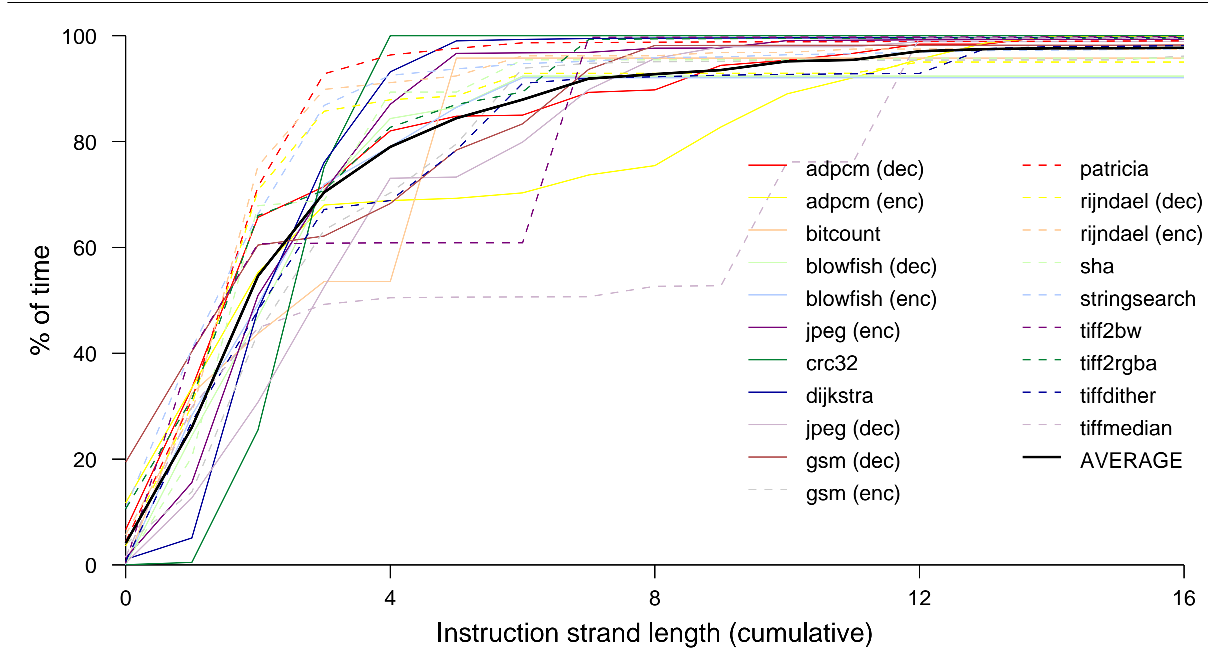
4.3.3 FIFO depth

The depth of the instruction FIFOs can affect performance in two ways. If the FIFOs are too small and fill, instruction issue will halt while they drain. If the FIFOs are too large, the additional complexity of the control logic and the greater wire delays from the larger structure could increase the cycle time and reduce overall performance.

It is important that a typical length instruction strand be able to fit within one of the instruction FIFOs, as all instructions from one strand must issue to the same FIFO. When the first instruction of a new strand is issued, it is steered to the emptiest FIFO, in order to distribute the load across all FIFOs. If a strand cannot fit within a FIFO, the issue logic will stall and later strands cannot be issued until enough instructions are executed from the FIFO in order to allow the remainder of the strand to issue, and thus instructions from later strands cannot be executed concurrently. Figure 4.11 shows the distribution of strand lengths in the executed code. These results show that over 90% of strands are eight or fewer instructions long, although some are actually much longer, up to thousands of instructions – further analysis shows that these long strands tend to be used as loop counters, and thus stay live over all iterations of the loop code (in cases where the inner code of the loop requires fewer than eight accumulators concurrently). If the instruction FIFOs are to be able to hold most instruction strands, then these results indicate that they should have at least eight entries.

Experiments were conducted to analyse the level of FIFO usage; the FIFOs were set to have unlimited capacity for these simulations. The simulator collected statistics on each FIFO, counting in how many cycles each level of occupancy occurred, and then averaging this data across each FIFO as a percentage of the simulation time. Figure 4.12 shows the distribution of FIFO utilization for the benchmark tests, for both 2-way and 4-way issue. The results show that there is a significant variation in the level of FIFO usage between the different benchmark tests; some, such as the *adpcm* tests, have very low FIFO usage, rarely exceeding eight entries, while other spend large portions of time with more than 20 instructions in their FIFOs. The simulations

Figure 4.11 Distribution of instruction strand lengths in ILDP code

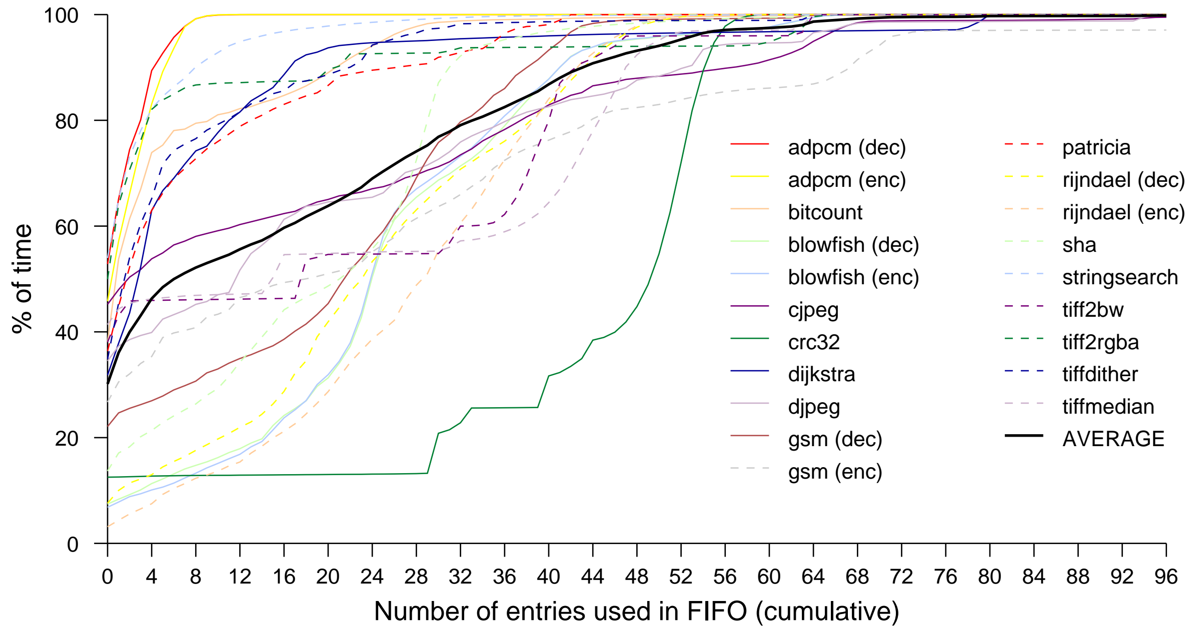


of the 2-way issue system show generally a much lower FIFO occupancy than the 4-way issue results – this is unsurprising, as the tests from the previous section have shown that typically the execution rate through the processing elements is not high enough to sustain 4-way issue, and as a result unexecuted instructions accumulate in the FIFOs; with 2-way issue the FIFOs are filled at a lower rate, closer to that at which they are drained.

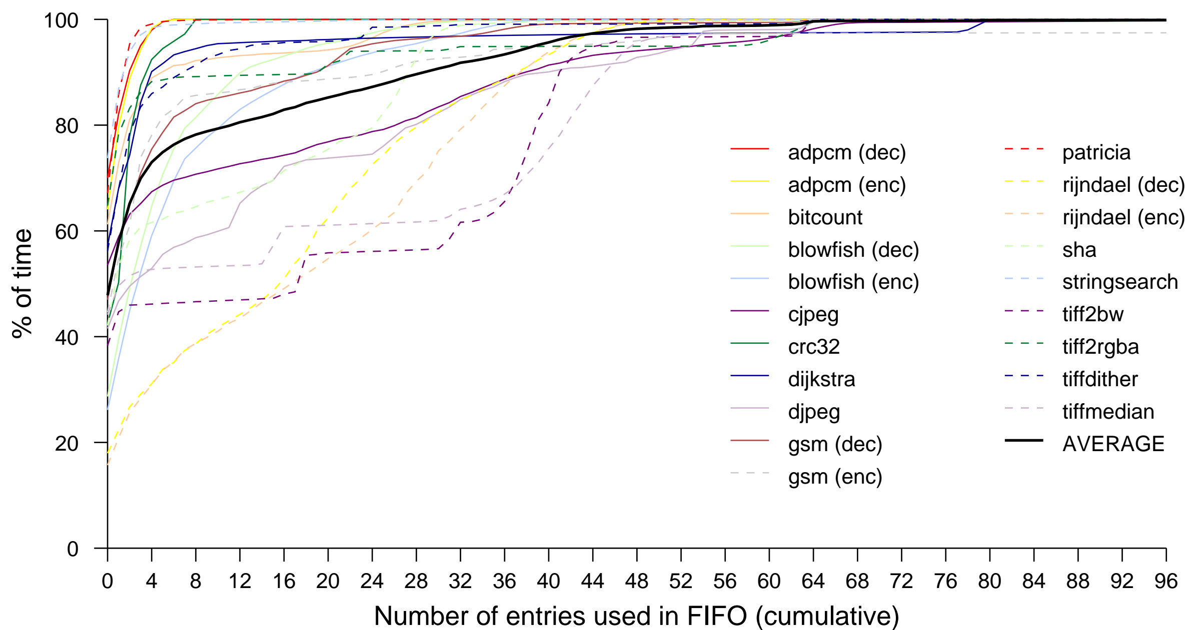
These results show that, for the majority of the time, there are fewer than 32 values in each instruction FIFO, particularly in the 2-way issue model. On average, for 90% of the time, the 2-way issue machine has a FIFO occupancy of 29 entries or less, while the 4-way issue results average 44 FIFO entries or less 90% of the time. From these results, it seems likely that a 32-entry FIFO will be necessary to avoid limiting performance – while the 4-way machine generally has a higher FIFO occupancy, a smaller FIFO may still not end up becoming a limiting factor. A smaller FIFO might still be sufficient – the important factor in the FIFO size is that a single PE stalling (on a load, etc) should not halt issue – instructions for that PE queue in its FIFO while other PEs continue executing. If all the FIFOs are filling up, it will not be a significant problem to halt the issue pipeline, although a buffer of queued instructions in the FIFOs can help mask latency caused by instruction cache misses etc. Further experiments were conducted, setting the FIFO size to 4, 8, 16, 32 and 48 entries. Figure 4.13 shows the performance obtained for these parameter settings.

These results show that there is very little overall effect on performance due to reducing the FIFOs from unlimited size all the way down to eight entries on both the 4-way and 2-way issue machines. Setting the FIFOs to a maximum depth of four entries causes performance to drop by 5.4% in the 4-way case and 8% with 2-way issue. This would suggest that it is sufficient to have the FIFOs sized to hold a single instruction strand – this allows later strands to issue afterwards

Figure 4.12 Instruction FIFO utilization

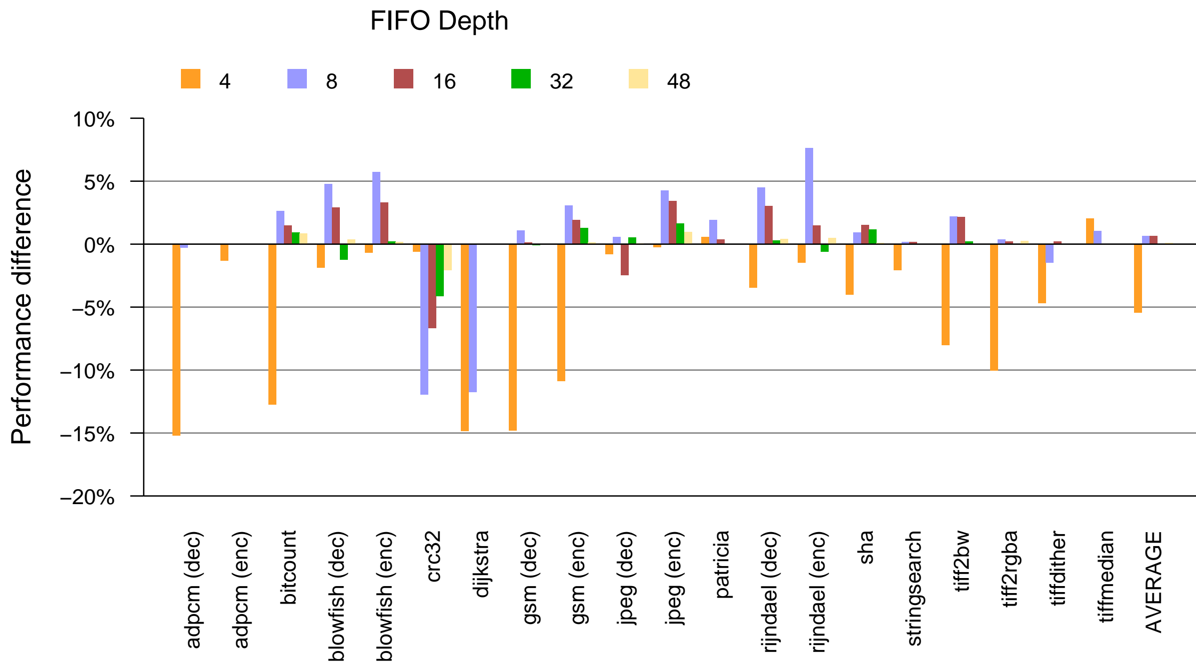


(a) 4-way issue

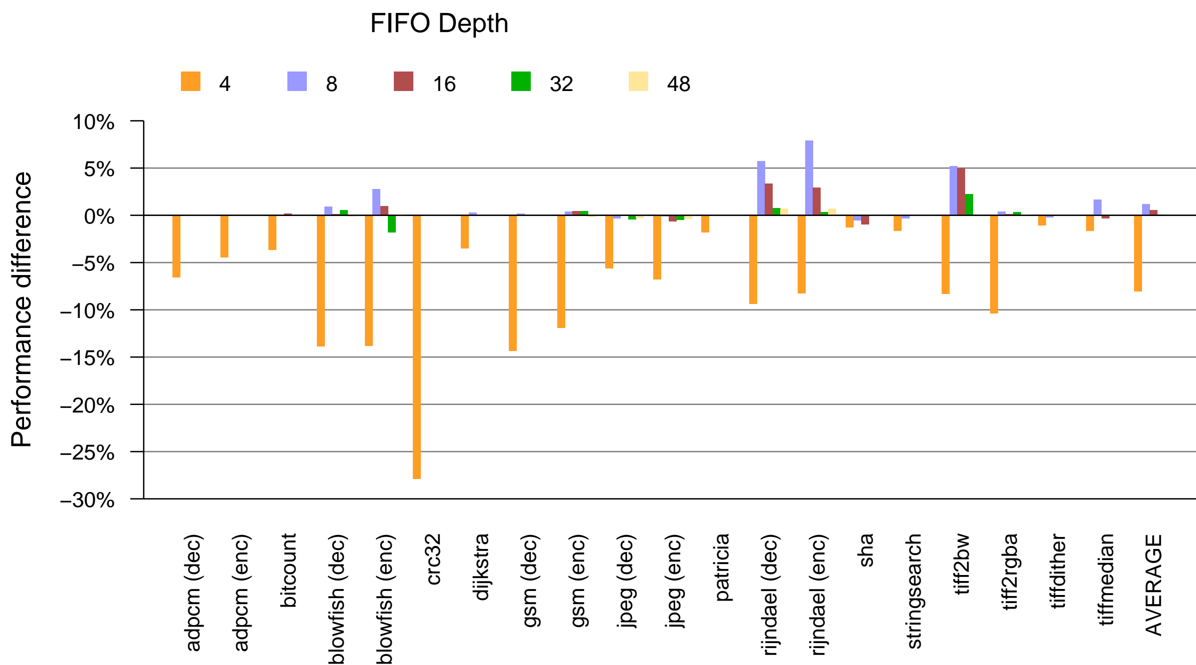


(b) 2-way issue

Figure 4.13 Performance at various instruction FIFO sizes, compared to an unbounded FIFO



(a) 4-way issue



(b) 2-way issue

but execute concurrently. Since there are eight instruction FIFOs, even for an issue width of four instructions per cycle, the extra room in a FIFO to hold two strands of eight instructions is not necessary. For shorter instruction strands (on average 79% of strands contain four or fewer instructions) an 8-entry FIFO will be able to hold two strands. As with the results in Section 4.3.1, some of the benchmarks actually perform better with a reduced FIFO depth, in particular for the 8-entry case.

4.3.4 Register network bandwidth

The number of values that can be simultaneously communicated over the register value network is an architectural parameter, which affects the complexity of the network logic and the register file logic – for a communication width n there must be n write ports to the register files in addition to any used by the PEs. If more values are produced in one cycle than can be communicated over the network then either some processing elements must be stalled until they can transmit their data, or excess values must be buffered.

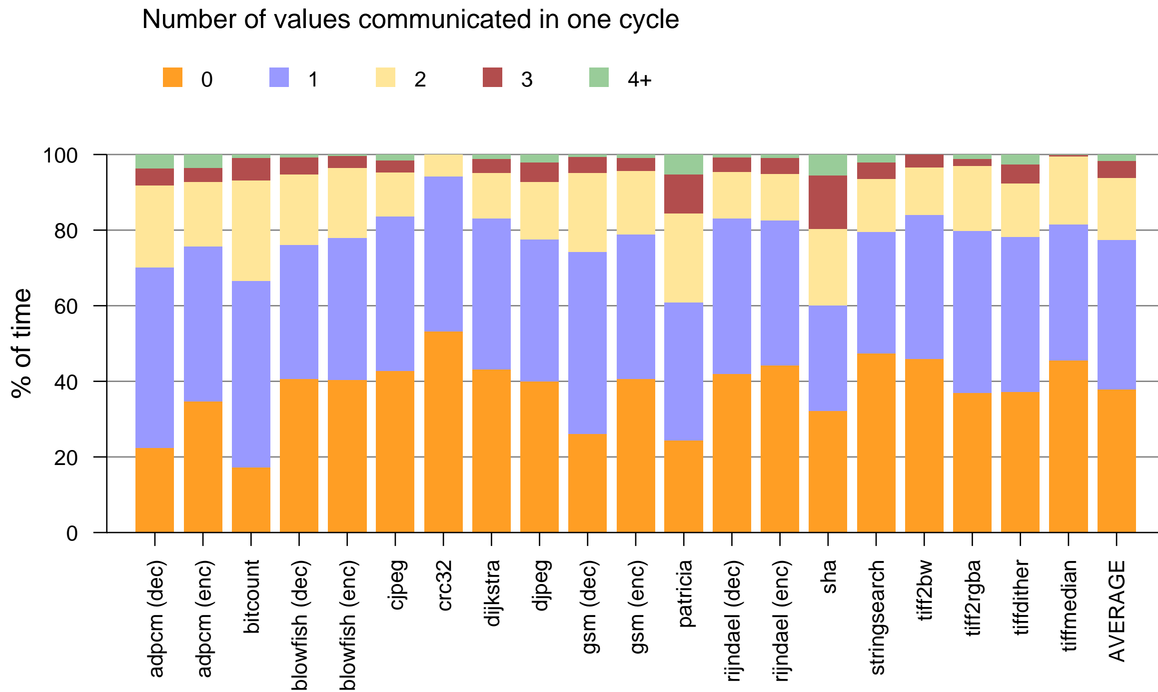
Initially the simulator was set up to allow an unlimited number of values to be broadcast within the same cycle, and tests were run to determine the average bandwidth utilization. Figure 4.14 shows the results of these tests.

These results show that, on average in the 4-way issue case, in 38% of cycles the register communication network is idle, carrying no data. In a further 40% of cycles a single value is transmitted over the network and in 15% two values are communicated. The 2-way issue results show a very similar distribution. Overall, for 77% of the time in the 4-way case and 81% of the time in the 2-way case, the register value network is carrying only one value or none at all. These results suggest that a register network bandwidth of one value per cycle will be sufficient to maintain performance, although this will depend on how critical to overall performance are any values that will be delayed by a bandwidth limitation.

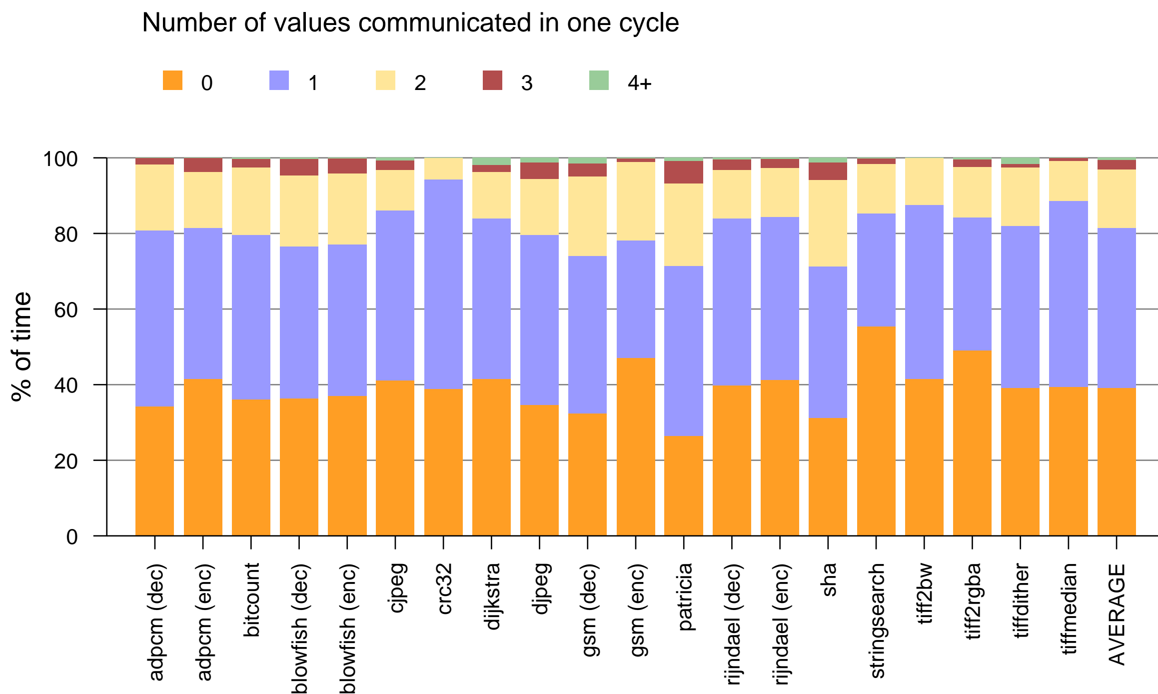
Further tests were then conducted, setting the maximum bandwidth to both one value and two values per cycle and evaluating the resultant impact on performance. The results are shown in Figure 4.15, with the performance figures normalised to the unlimited bandwidth simulations.

These results show that the setting the register network bandwidth at two instructions per cycle has very little effect on performance compared to having unlimited bandwidth – this is unsurprising, as in the earlier tests more than two values were communicated in a single cycle for less than 4% of the simulation time in the 2-way case, and less than 7% in the 4-way case. The single-value bandwidth results show a performance reduction of approximately 3% on the 4-way issue tests, indicating that delaying some register updates by a cycle when there is contention on the communication network does not have a critical influence on the overall performance. In the 2-way issue tests, the single-value case performs less than 1.4% worse than

Figure 4.14 Register network bandwidth utilization

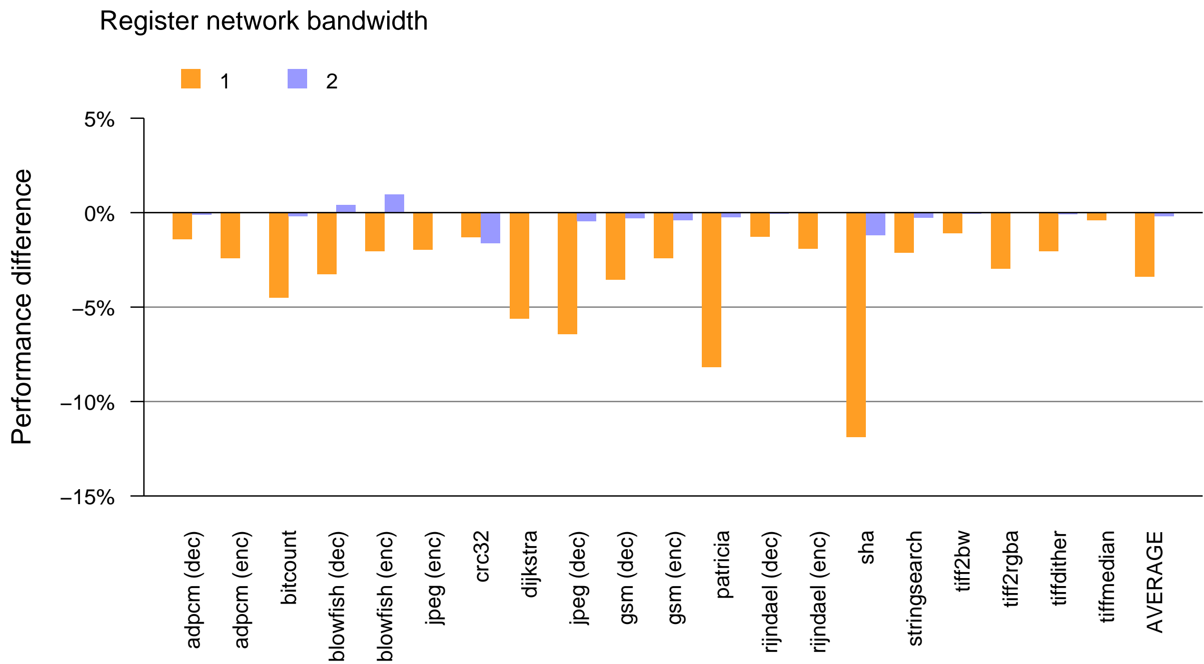


(a) 4-way issue

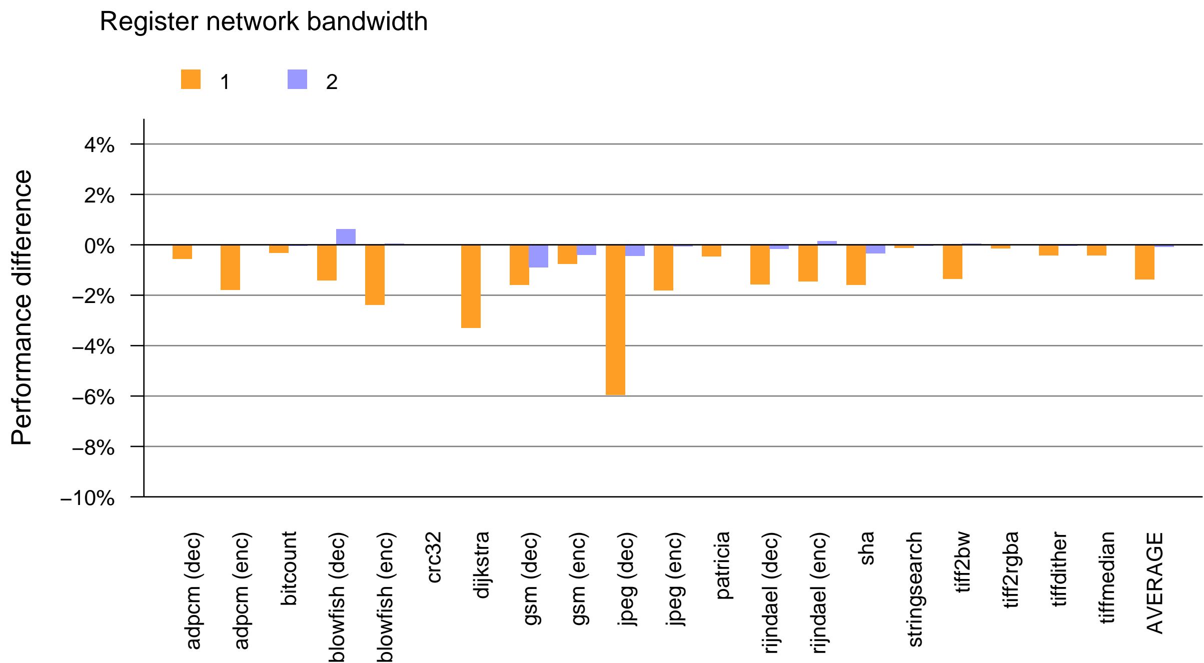


(b) 2-way issue

Figure 4.15 Performance at various register network bandwidths, compared to unbounded case



(a) 4-way issue



(b) 2-way issue

the unlimited case, indicating that for 2-way issue, a single-value register network is likely to be sufficient.

Overall, given the added complexity of implementing register communication logic capable of carrying two values in one cycle, which will require an additional register write port, it seems likely that the best performance/complexity trade-off is to limit the register network bandwidth to a single value per cycle.

4.3.5 Physical register file size

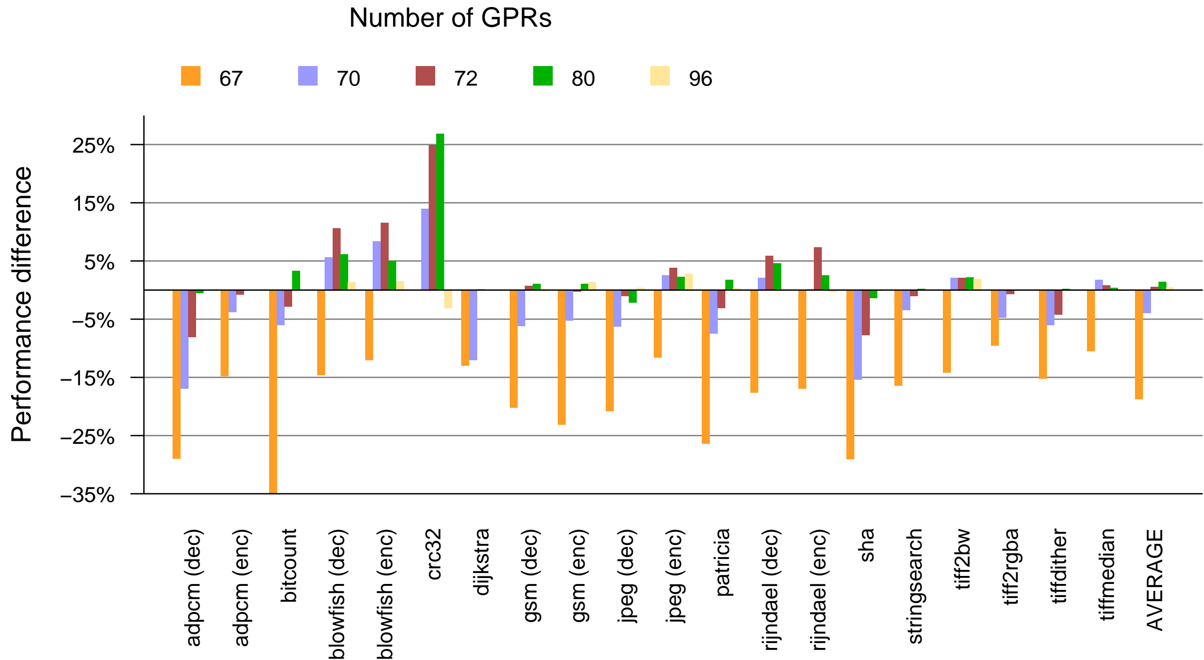
Experiments were conducted to see the effect of varying the size of the register file in each of the processing elements – in the reference ILDP architecture the 64 GPRs are renamed onto a register file with 128 entries. A smaller register file consumes less power and has a smaller logic delay ([42] shows that register file delay scales linearly with the number of registers), but reduces the number of possible in-flight instructions and can therefore reduce performance.

Simulations were conducted with a register file size of 80 and 96, in addition to the baseline model with 128 registers. After analysing the results from these simulations, very little performance variation between 128, 96 and 80 registers is displayed. It was decided to run a further set of tests was run with the size set to 72, 70 and 67 physical registers (due to the implementation of the simulator, there must be at least three more physical registers than logical ones, thus 67 is the minimum). The results from these simulations are shown in Figure 4.16.

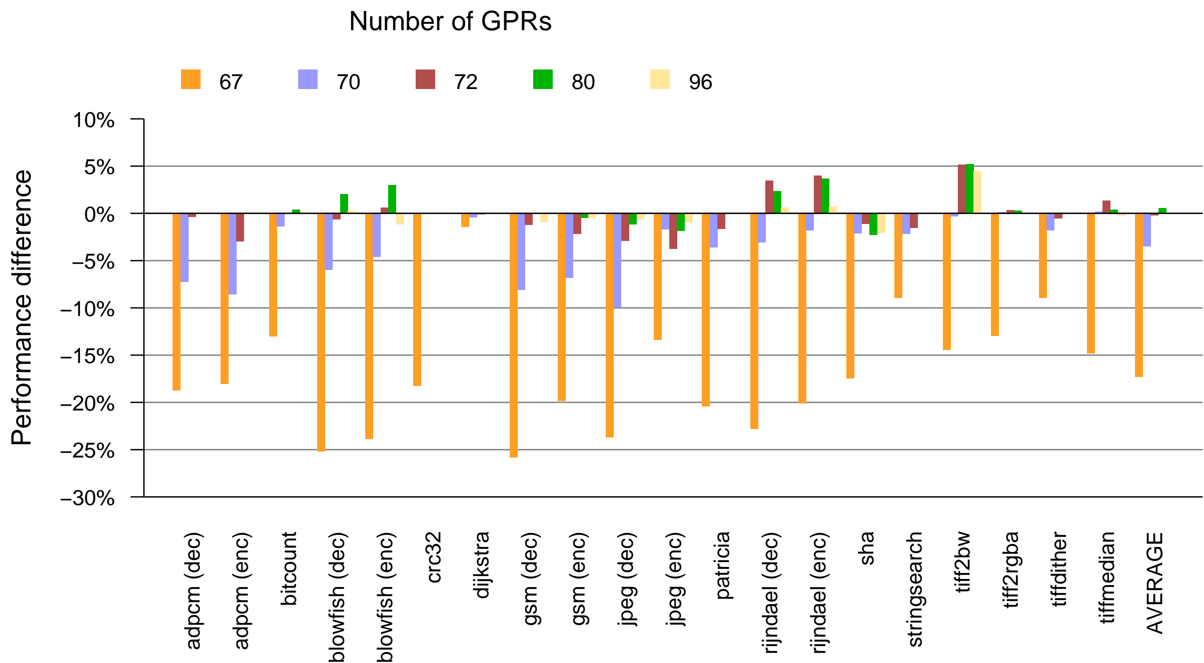
Once again, the *crc32* benchmark gives anomalous results in the 4-way issue case – reducing the number of physical registers to 80 results in a 25% performance increase. As in Section 4.3.1, this result is taken as an outlier, and not counted in the average.

On average, there is very little performance difference between the tests with a 72-entry register file and those with a 128-entry file. Performance drops on average by 4% when the register file size is reduced to 70 entries, and the minimal 67-entry case performs 18.7% worse than the baseline 128 register model. This implies that there are generally very few live registers at one time within typical workloads – probably often between six and eight, given the performance difference between the 70-entry and 72-entry cases. This is attributed to the accumulator-based ISA – most values are communicated locally in the accumulators, reducing the level of usage of the general-purpose register file. Given this, it seems likely that the register file size could be reduced considerably by also reducing the logical register file size exposed in the ISA, which is investigated in the next section.

Figure 4.16 Effect of scaling physical register file size from 128 GPRs on performance



(a) 4-way issue



(b) 2-way

4.3.6 ISA register file size

There is a limit to how much the physical register file can be scaled, as it must still contain all of the logical registers, even if the majority of them are dead. If too many logical registers are specified in the ISA then those registers that are dead for most (or all) of the time will waste space in the physical register file, and possibly limit performance as the register rename map will fill sooner. Too few logical registers will force unnecessary stack spills to be inserted in the instruction stream, limiting performance due to the additional instructions. It is therefore important to find an appropriate register file size, sufficient to hold most of the live data in most cases, but small enough to ease implementation.

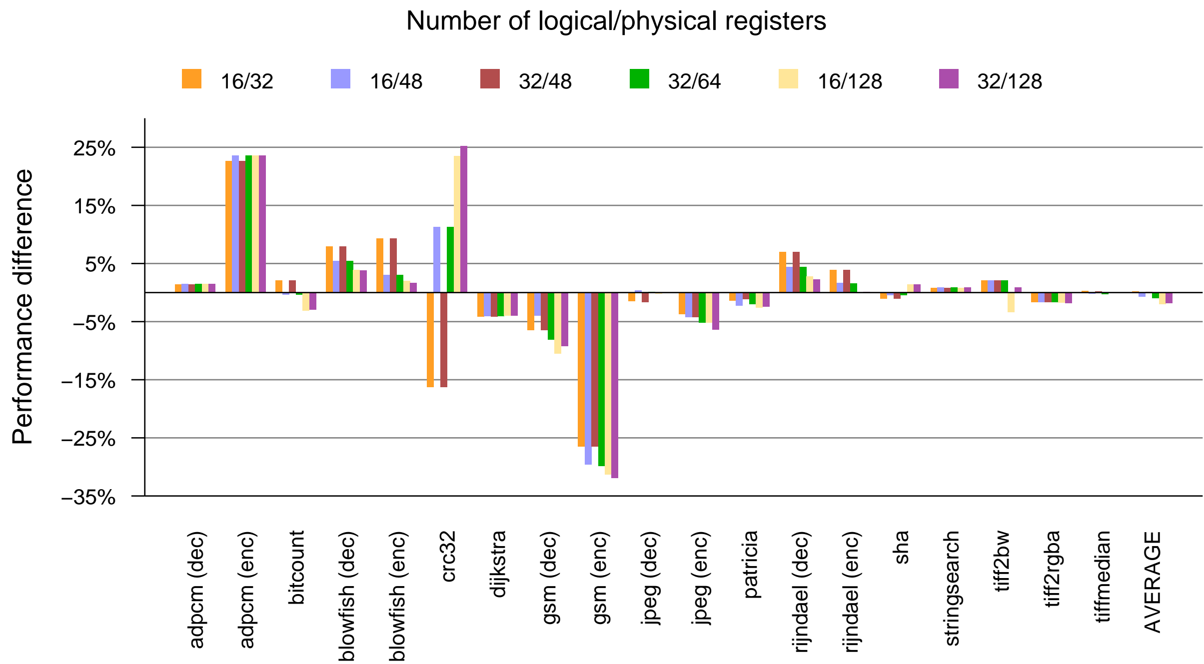
The ILDP ISA specifies 64 GPRs – this is due to the design in [30] being targeted at executing Alpha binaries via dynamic translation. The Alpha ISA contains 32 GPRs which are mapped to ILDP registers r0–r31, and an additional 32 registers are provided for the use of the binary translation software. In my evaluations, without the additional register requirements of the dynamic binary translation system, it seems likely that fewer logical registers will be sufficient. Having fewer logical registers also reduces the complexity of the register rename logic slightly.

Experiments were thus conducted varying the architectural register file size to see if it could be reduced without significant impact on performance – finding the optimum trade-off between having enough architectural registers to hold live values and avoid stack spills, but not so many that the physical register file is filled with dead values or the cycle time is compromised. The compilation system was modified to generate code using only a subset of either 16 or 32 of the GPRs, and the simulator was modified so that only these subsets were actually implemented. Figure 4.17 shows the results of these tests, with several sizes of both the logical and physical register files evaluated. There were some problems getting the *tiffdither* test to compile with the reduced register file – this test has been omitted from these results.

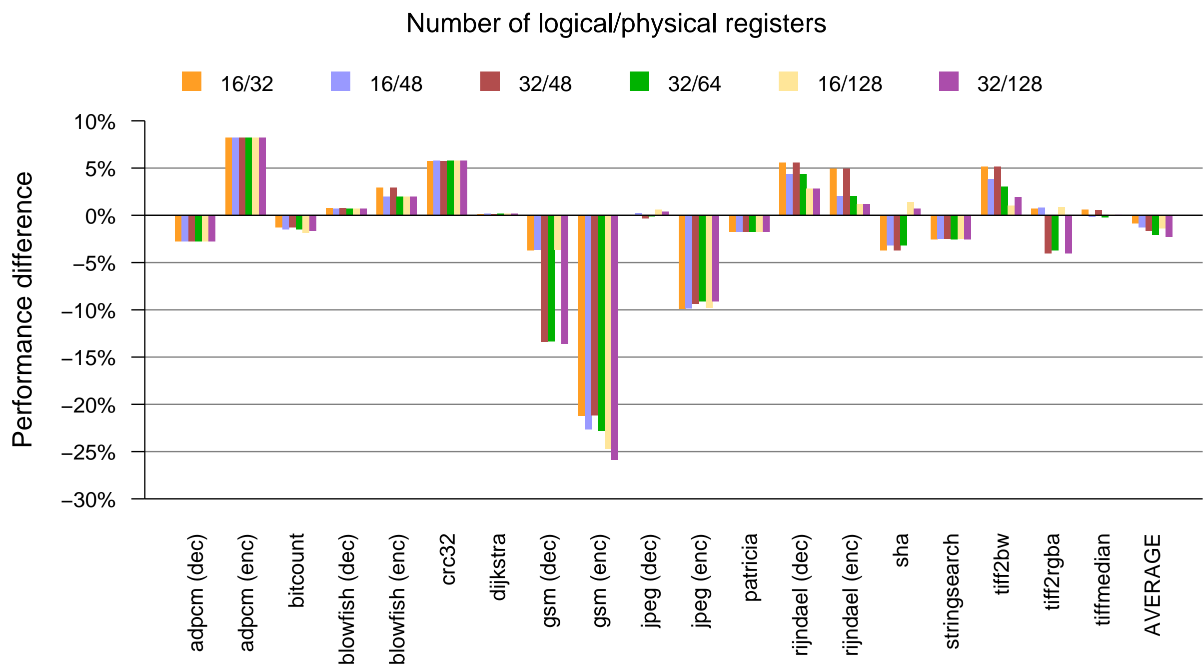
It can be seen that for most tests there is little performance difference with the reduced register files. The most notable exception to this is the *gsm (enc)* benchmark, which consistently performs more than 20% worse in each of the tested configurations. Analysis of the simulation statistics show that for this test, 21% more instructions are being committed in total – while the IPC drop for this test is not overly severe, the added spill instructions generated by the compiler due to the reduced register file cause a significant performance hit. The *adpcm* test shows a significant performance increase in all the tested configurations – this is due to a higher branch prediction accuracy than the baseline results, though it is not obvious why this is the case; it is assumed that the baseline test contains a sequence of code that is badly predicted by the branch prediction scheme, and the reduction of the register file size changes this code sufficiently to avoid this. In the 4-way configuration, the *crc32* test once again shows a large performance improvement when the logical register file size is reduced.

Both the tests with 32 logical registers and 48 physical registers and with 16 logical registers and 32 physical registers provide comparable performance to the base 64-logical, 128-physical

Figure 4.17 Effect of scaling both physical and logical register file size on performance



(a) 4-way issue



(b) 2-way issue

case on average, and should allow for a great reduction in the size and complexity of the register file and rename logic. Some tests suffer from the modified ISA – it is possible that this could be at least partially rectified through compiler optimizations, but ultimately the specific workload an application would be using will dictate whether reducing the register file size is beneficial for a particular design.

A further potential benefit of a smaller logical register file, which is not explored here, is that fewer bits are required in the instruction encoding in order to specify the register number for register operands, allowing more bits to be used on the opcode or immediate fields. This would make the instruction encoding more efficient and hopefully allow for a reduction in code size.

4.3.7 Cache parameters

High performance processors with deep pipelines and short cycle times generally have a first-level cache with low associativity to reduce circuit complexity so that the cache can meet the timing requirements with a single-cycle latency. By contrast, embedded processors are generally less aggressively pipelined and tend to have longer cycle times. In these conditions, first-level caches can be made more associative, improving their hit rates.

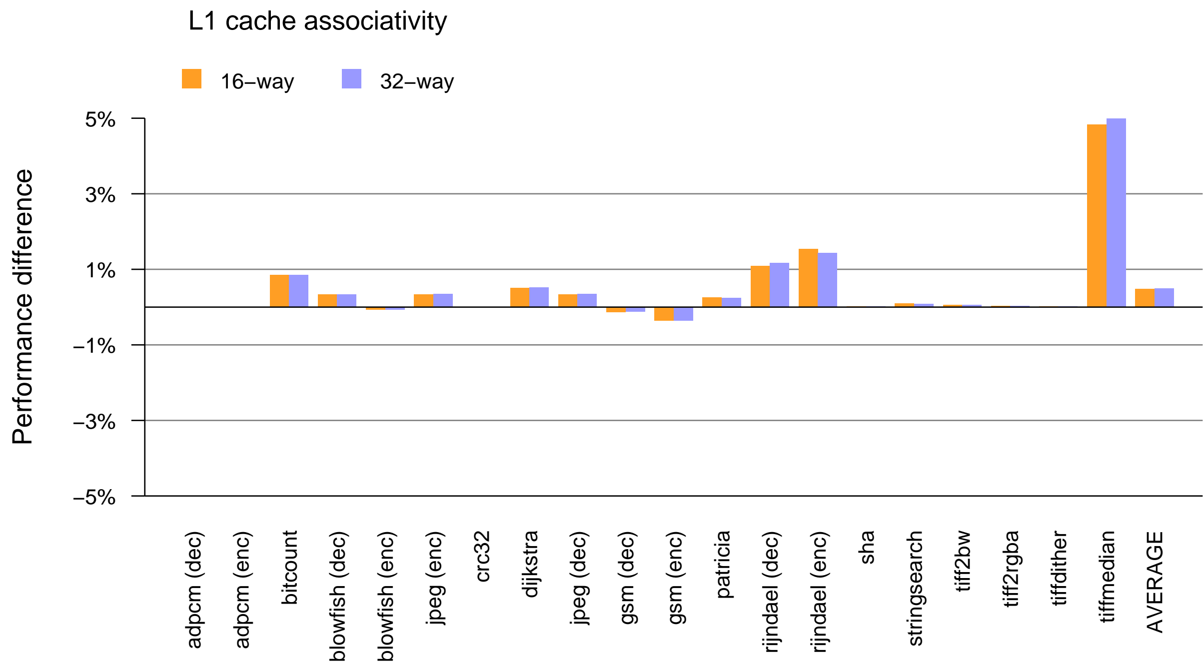
Since the work in [30] is targeted at high performance, low cycle time design, the baseline ILDP architecture uses an 8KB 2-way set associative cache. I evaluated performance of the ILDP architecture with the cache associativity increased to both 16 and 32 ways, to analyse how sensitive the performance is to cache associativity.

Figure 4.18 shows these results. It can be seen that across most of the tests there is very little performance variation when the associativity is increased – only the *rijndael* and *tiffmedian* tests show more than a 1% difference. These results would seem to indicate that, for this set of benchmarks, a 2-way associativity on the cache is sufficient in the ILDP architecture. This is attributed to the characteristics of the MiBench benchmark suite – most tests have a fairly low memory footprint, and thus tend to get high hit rates to the L1 cache even with the restricted parameters present; increasing the associativity thus has little effect.

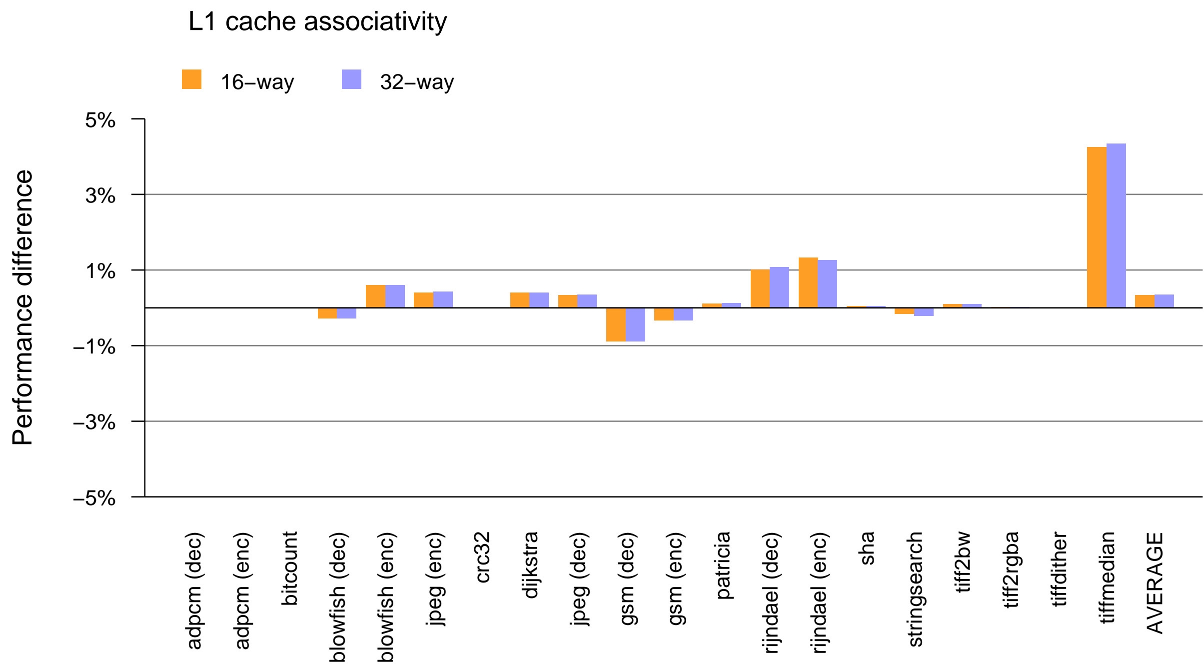
4.4 Combining approaches

In the previous section various architectural parameters were identified and analysed, and then simulations were run to evaluate how each of these parameters affect performance when altered in isolation. In this section results are presented when those parameters are set to the values likely to reduce circuit area and power consumption without significantly impacting upon performance.

Figure 4.18 Effect of increasing cache associativity from 2-way on performance



(a) 4-way issue



(b) 2-way issue

Figure 4.19 Effect of varying several architectural parameters

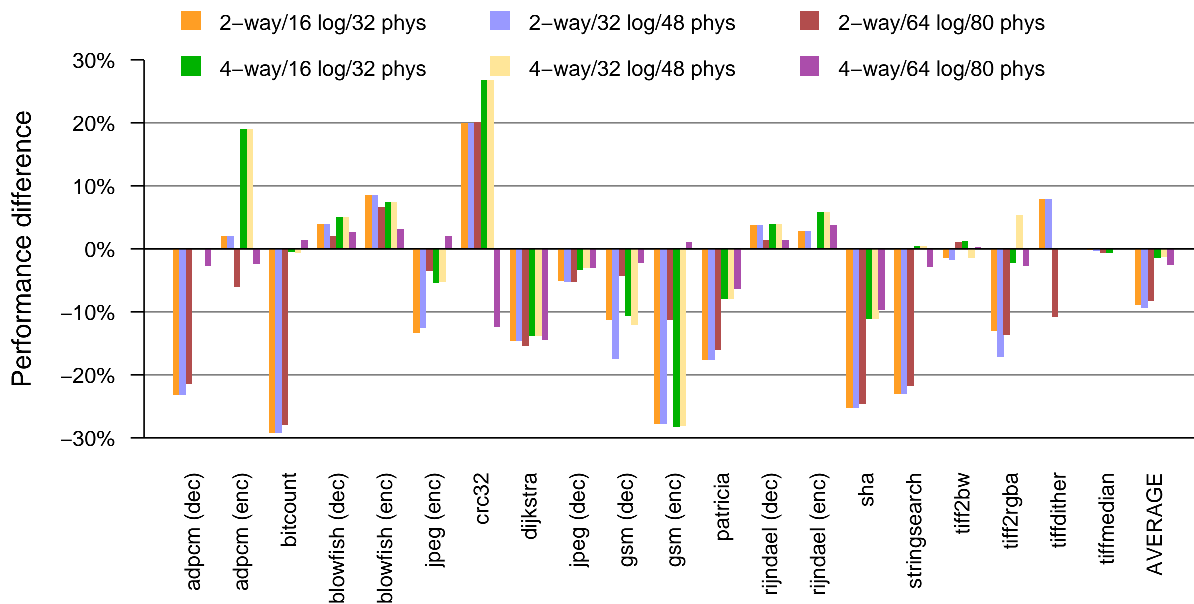


Figure 4.19 shows the results of combining several sets of architectural parameters. Several configurations of register file size were tested at both 2-way and 4-way issue widths, with the maximum FIFO depth set at 8 entries and the bandwidth of the register value communication network set at one value per cycle. The results obtained are as expected – they reflect closely the results obtained earlier when the parameters were investigated in isolation, indicating that no unanticipated performance changes arise when multiple paramters are altered. The next section discusses suitable final choices for values of each of the tested parameters.

4.5 Summary

In this chapter, the ILDP architecture has been analysed compared to several typical embedded processors, and shows good, but not exceptional performance. However, the overall IPC of ILDP is more than double that of the other architectures investigated, and shows scope for considerable performance improvements from compiler improvements.

The architectural parameters set for this architecture were also explored; it is shown that scaling the issue width from 4-way to 2-way has the potential to reduce logic complexity, but comes at a 9% performance penalty that may be too great for certain applications. The best trade-off for the instruction FIFO size was found to be eight entries and the register network performs best when it can communicate two values concurrently, but a reduction to a bandwidth of one does not significantly impact upon performance, giving a 3% drop for the 4-way issue model. The 128-register GPR file and 64-register ISA were found to be excessively large for the test suite

used; a logical register file of 16 entries and a 32-register physical register file were found to provide performance within 3% of the original model in most cases.

A further observation that can be made from the data presented in this chapter is that increasing the parallelism of certain components within a processor can, without considering the effect on cycle time, actually decrease performance per clock. In particular, if the issue pipeline is able to run too far ahead of the execution logic, performance can suffer as the chances of misprediction or cache pollution increase. This highlights the need to balance the parallelism of the various components within a design.

5 Architectural developments

This chapter details new developments on the ILDP architecture in order to make it more suited to embedded systems.

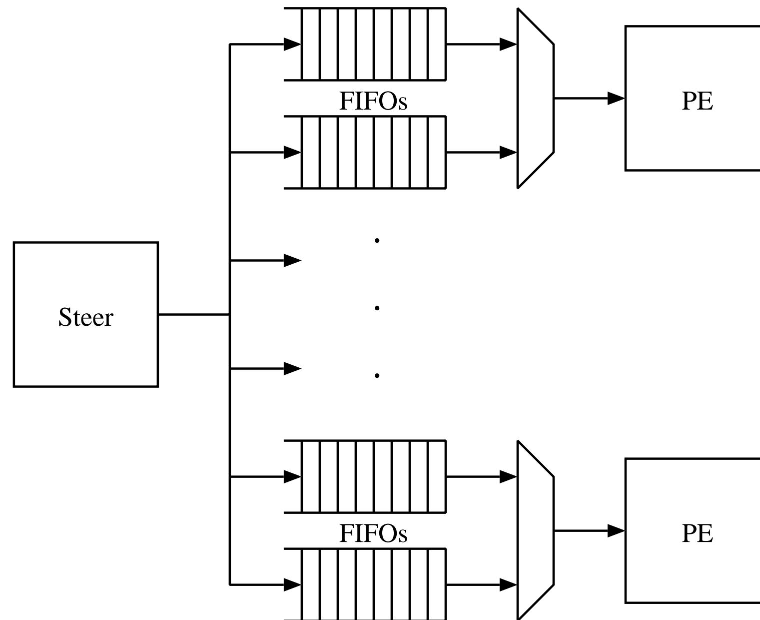
5.1 Multiplexing instruction FIFOs onto a processing element

As is shown in section 4.2.2, the processing elements spend most of their time idle, either because their instruction FIFOs are empty or because they are stalled on a memory access or register value. While inactive PEs can potentially be clock gated or powered down, those waiting on data or memory dependencies must remain at least partially powered up. Even when a PE is powered down, it must either maintain and update the contents of its register file by keeping the register file and update logic powered, or it must synchronise the contents of its register file on power-up, increasing the latency of restarting a PE and potentially negating the power savings achieved by powering the PE down. This indicates that power savings could be made by reducing the number of PEs – since each instruction queue is stalled most of the time, multiple queues can be assigned to each PE, which can interleave execution of the two with hopefully very little performance penalty.

This section evaluates the potential of this approach. Since the PEs spend slightly more than half the execution time in an idle state, a FIFO:PE ratio of 2:1 seems like the best option to reduce hardware requirements without unduly affecting performance. This ratio would halve the number of register files and ALUs on-chip, and reduce the memory access requirements – either half the number of L1 caches would be required, or each L1 cache would need half as many ports.

Since there are half the number of PEs, the peak execution width is also reduced by half, but it is hoped that this will not significantly impact performance (the average execution rate is much lower than the theoretical maximum execution rate) and that the reduction in total silicon area – and thus power consumption – will outweigh any degradation. If half the PEs are eliminated whilst maintaining performance, this should greatly improve the power efficiency by better utilising the remaining logic.

Figure 5.1 Multiplexing instruction FIFOs between processing elements



5.1.1 Implementation details

The basic structure of the multiplexed FIFO architecture is shown in Figure 5.1. The front-end issue pipeline does not need to be changed in order to implement multiplexed processing elements; instructions are still issued to the tails of the eight FIFOs, although the steering logic should probably be tweaked so that new instruction strands are preferentially steered to PEs where *both* FIFOs are empty.

Each processing element is modified by adding a second accumulator register and FIFO. Multiplexers are added to select between the two FIFOs and accumulators. Some logic is required in order to decide which FIFO to execute from – for this a second set of dependency logic is required, to indicate if the head instruction of the additional FIFO is ready for execution. If only one of the instructions from the FIFO heads is ready, the arbitration logic selects that instruction for execution. If both are ready, the instruction is taken from the opposite FIFO to that last used – a single bit of storage is required to indicate which FIFO was previously executed from.

In terms of complexity, the multiplexers on the outputs from the FIFOs and the accumulators will add some additional delay in the PEs, which could potentially increase the critical path. The additional logic will also increase the silicon area of each PE somewhat, although the area benefits of halving the number of PEs will greatly outweigh this small increase.

5.1.2 Evaluation

The simulation environment was modified to support sharing a processing element between multiple instruction FIFOs. The processing element has one accumulator per FIFO. If only one of the instructions at the FIFO heads is ready for execution, then it will be issued. If more than one is ready, the instructions are issued in a round-robin fashion. Experiments were conducted reducing the number of PEs to four, with each PE having two instruction FIFOs.

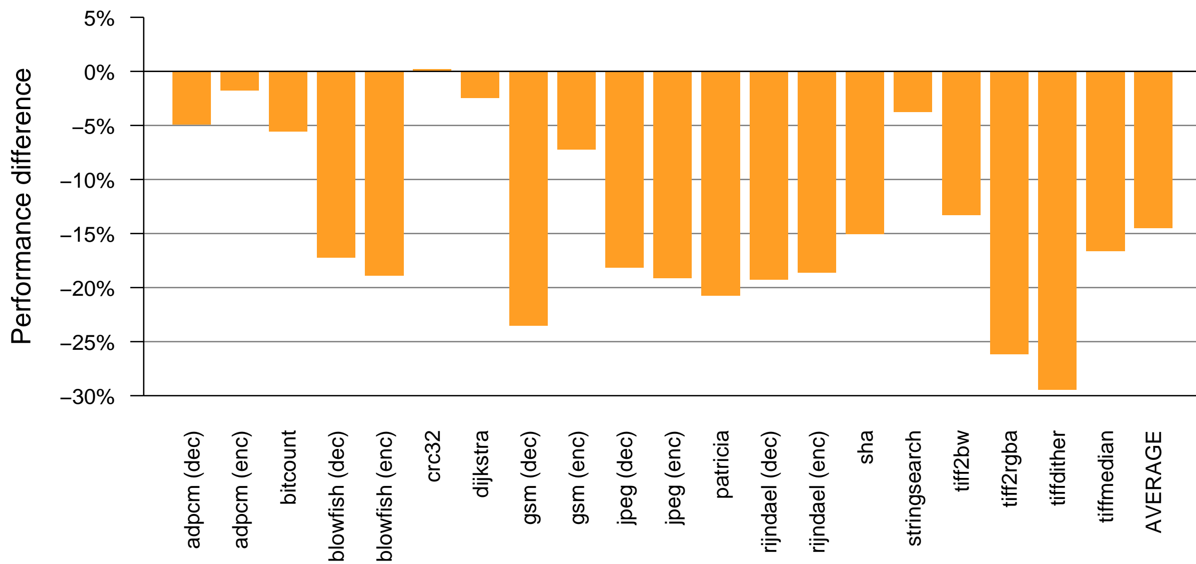
The results in Figure 5.2 show that there is a significant performance hit using this configuration – on average the 4-way case is 14.5% slower, while the 2-way results average 13.1% worse. It seems that, while overall the PEs are idle most of the time, the access pattern is ‘bursty’ and the full execution width is required to meet the peak performance requirements. In the 2-way case, the best result (the *adpcm (dec)* test) is only 1.5% slower, while with 4-way issue the best result (*adpcm (enc)*) is 7.2% worse. This is attributed to the lower peak issue rate of the 2-way issue model – with a lower issue rate, the reduced peak execution rate will be a limiting factor less often.

The utilization statistics for the processing elements in the multiplexed FIFO model are shown in Figure 5.3. The average time spent executing instructions has increased from approximately 21% to 36%, showing that better overall utilization is being made of the execution logic.

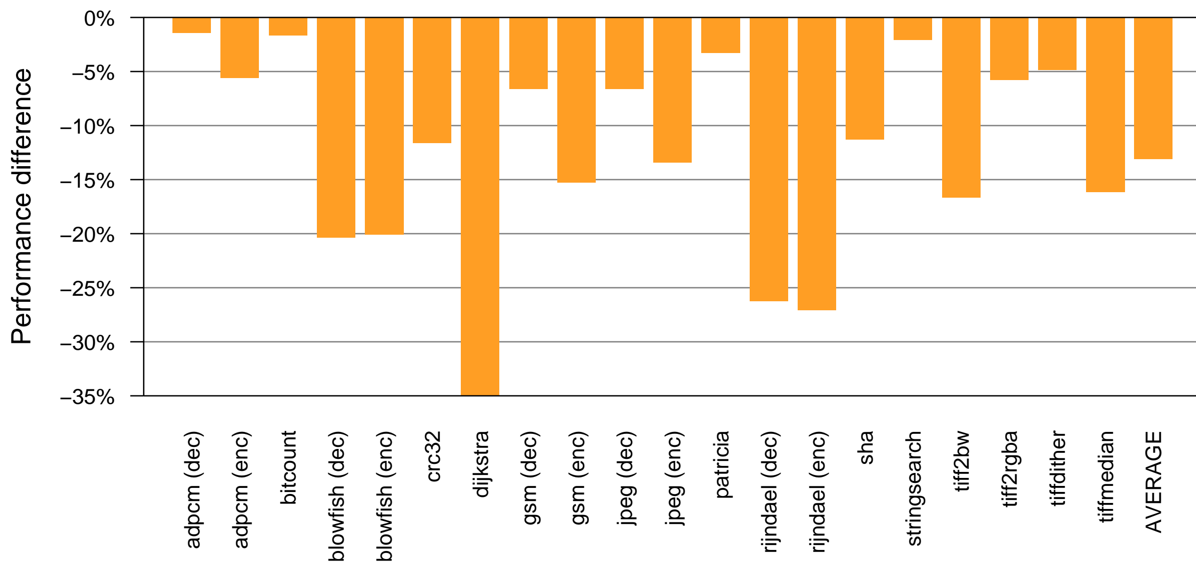
5.1.3 Conclusions

Multiplexing FIFOs between processing elements allows for large area savings – only half the number of execution units, register files and memory access ports are required. There will also be potentially large power savings – even if PEs can be extensively clock and power gated, the register files (and associated synchronization logic) must be powered up at all times consuming power, and halving the number of these will reduce power consumption. If clock/power gating is not used, then halving the number of PEs will give an even more significant reduction in overall power usage. These savings come at the cost of a significant 12–14% performance impact (although the reduction in the overall complexity of the memory access logic could potentially reduce the cycle time and slightly mitigate the performance drop), which may well prove too great a cost for the power and area savings achieved. Section 5.3 presents an alternative method of reducing the power consumed by the register file logic that limits the effect of power gating.

Figure 5.2 Effect on performance of multiplexing instruction FIFOs between PEs

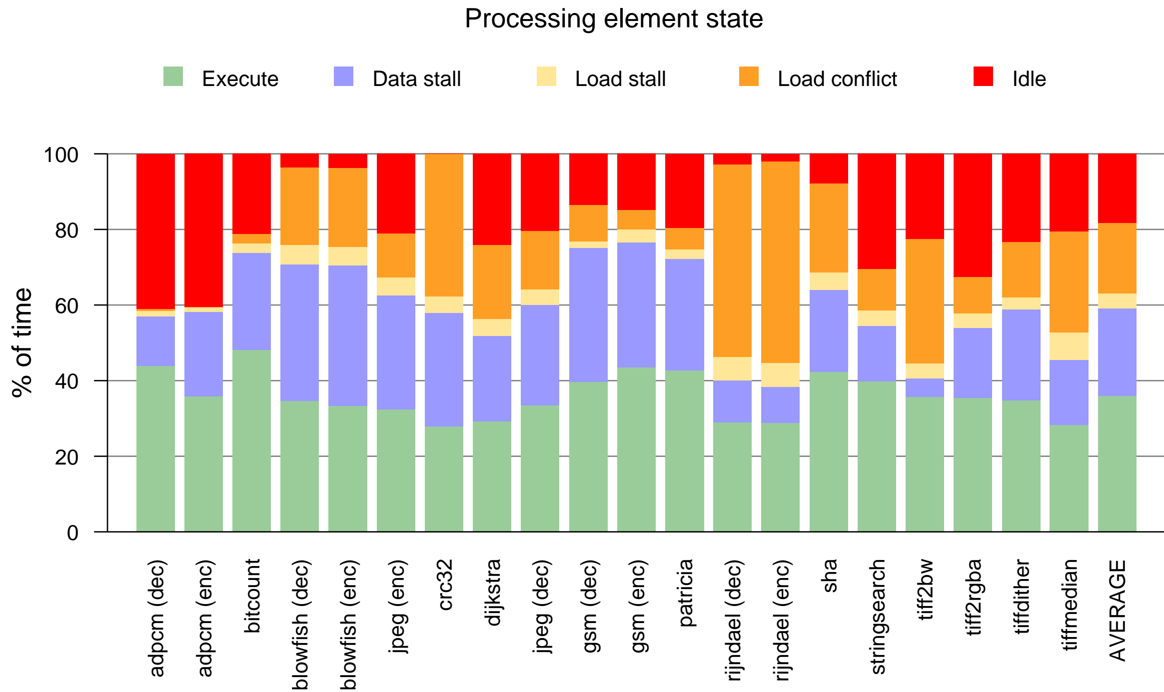


(a) 4-way issue

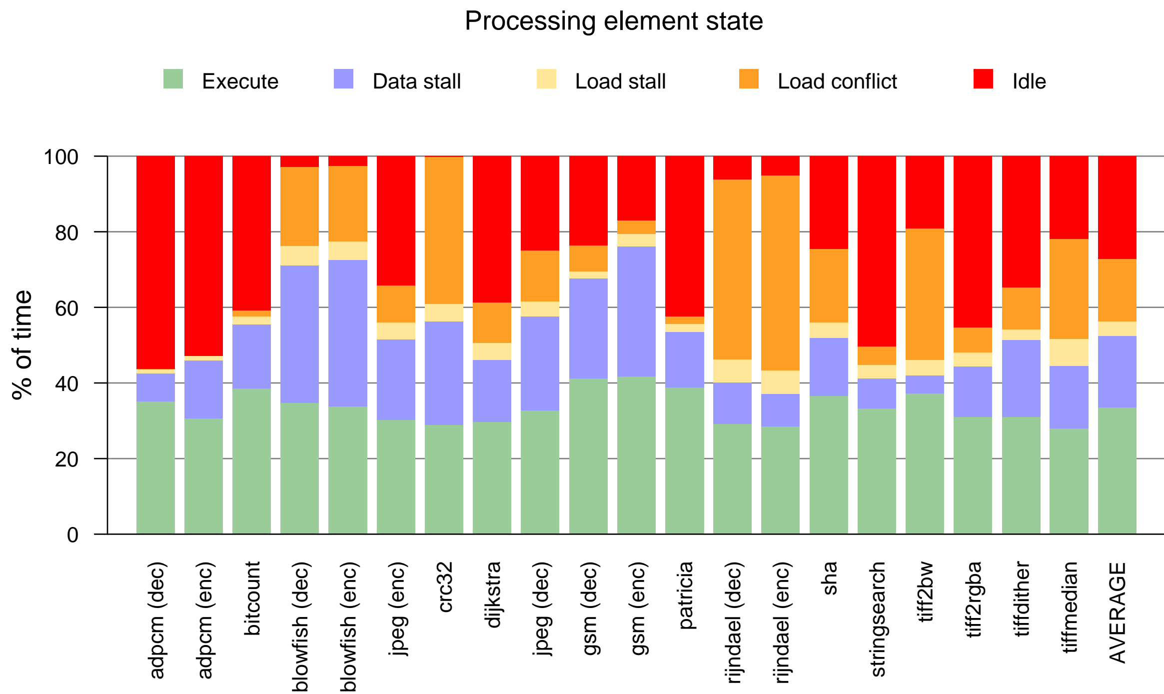


(b) 2-way issue

Figure 5.3 Processing element utilization with two FIFOs per PE (compare to Figure 4.5)



(a) 4-way issue



(b) 2-way issue

5.2 Combining memory access units

The baseline ILDP architecture has a memory port for each PE – this requires either replicating a single-ported L1 data cache across each PE, or implementing multi-ported caches which are shared between PEs. Neither of these options is particularly space- or power-efficient, as the caches are configured to have identical contents replicated between them, so the silicon area and power used by the additional caches is effectively wasted – a configuration with four dual-ported caches will be wasting 75% of the L1 cache silicon area and using four times as much power as a single cache.

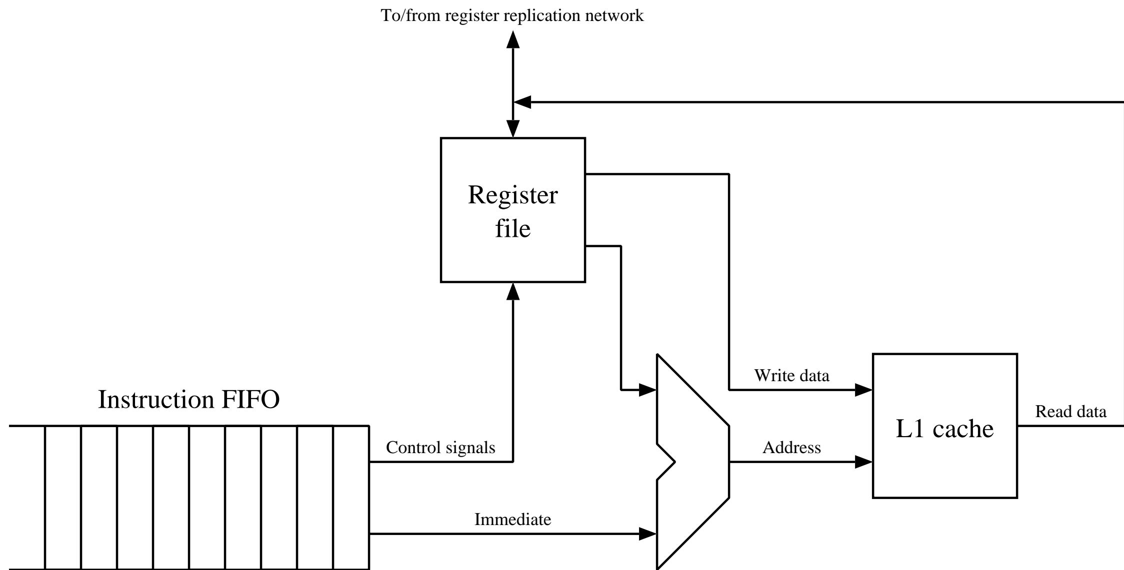
To maintain ordering of memory references and prevent loads from overtaking earlier stores, each PE must contain a copy of the store address queue. When a load executes, it checks the queue for any earlier stores to the same address and, if these or any earlier store instructions with unknown addresses are present, the load will stall. In order to keep the store address queues synchronised between PEs, a second set of replication logic similar to the register file replication logic is needed. As with the register file logic, if a PE is powered down, the store address queue, cache and associated logic must remain powered up in order to keep their state synchronised with the other PEs.

This section proposes a development on the ILDP architecture where memory accesses are handled by an additional processing element solely for memory access, called the *memory access unit* (MAU). All memory access instructions in the ISA are modified to operate on general purpose register operands only, instead of using any of the accumulator registers. The structure of this unit is shown in Figure 5.4; the MAU contains a simple ALU for performing address computation, and the sole copy of the L1 cache. This modification removes the need for replication of the cache (and the associated logic for maintaining cache coherency) and the replication network for synchronising the store address queues between PEs. There is no need for a store address queue in the MAU, as all memory instructions are naturally ordered in the instruction FIFO.

There is the potential for performance issues to arise from the ISA modifications – since memory access instructions do not use accumulator registers, they can no longer form part of strands and thus the average strand length will possibly be shorter, which may hamper performance as a higher proportion of the instructions will be used to transfer data between PEs and the register file than to actually perform computation. There are some cases which benefit from the changes though – an example is restoring GPR values from the stack, e.g. at the end of a function call, as the original ISA required registers to be loaded into accumulators and then moved into GPRs.

This approach could allow a more sophisticated issue system to be used for the memory access unit than the main logic – a full out-of-order issue window could be implemented, with the reduced size of the window hopefully making the complexity/performance trade-off acceptable. This would more easily allow load instructions to be hoisted ahead of non-conflicting stores. Since memory operations are more likely to cause long delays, they will benefit more from

Figure 5.4 Design of memory access unit



an advanced issue system than the main processing elements. This approach is investigated in Section 5.2.4.

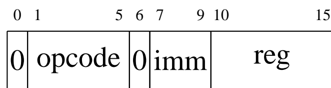
5.2.1 ISA modifications

The load and store instructions are modified to access only GPR operands. The encoding for these instructions was chosen so that the stack pointer (r62 in the chosen ABI) can be specified as the base register using fewer bits, allowing more bits to be allocated to the immediate offset. The new instructions are described in Table 5.1. Since these instructions have two register operands, they will require alterations to the register rename logic – either additional ports will have to be added to the rename map, or renaming of a memory instruction will cause a drop in issue bandwidth. It may be possible to implement the stack pointer as a special case in these circumstances, mitigating the penalty for the common case of memory operations using the stack pointer as the base register.

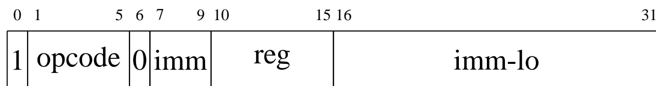
Syntax	Semantics
Load/store instructions	
`ldr r<i> <= [r<j>+#imm]`	$R_i := MEM[R_j + imm]$
`str [r<j>+#imm] <= r<i>`	$MEM[R_j + imm] := R_i$

Table 5.1: Instruction set modifications

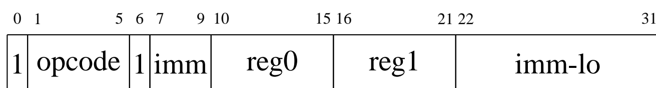
Figure 5.5 Additional instruction encodings



Short load/store format (stack pointer is base register)



Long stack-based load/store format



Long register-based load/store format

5.2.2 Implementation details

For this approach, the issue logic is modified to issue all memory operations to the MAU – all other instructions are issued as normal to the standard PEs. The processing elements have their memory access logic eliminated, along with their copies of the store queue. The cache coherency logic and the communication network for updating store queue entries between PEs are also removed.

The register rename logic must be changed to handle the new memory instructions, which require either reading two GPRs, or reading one and writing one. This necessitates either adding more ports to the rename map to increase the peak rename bandwidth, or keeping the same number of read ports and stalling some instructions when memory instructions are renamed. Both implementations are investigated.

The new memory access unit contains an instruction FIFO and a copy of the register file like the other PEs. There is a simple ALU (basically just an adder) to perform address computation, the L1 data cache and the write buffer. Load instructions can read data not yet written to the cache from the write buffer.

In terms of complexity, in this design the main processing elements become slightly simpler as the memory logic is removed. The memory PE instead has this logic, but without the need for load/store dependency checking as memory operations are issued in order. The adder in the memory PE will have lower complexity than the full ALU in each PE. Eliminating replicated caches and/or reducing the number of memory ports per cache will reduce total silicon area usage and hopefully also cache complexity – even if the caches were originally single-ported, the cache replication logic can still be eliminated. There is likely to be a (modest) reduction in the overall delay through the processing elements, which, if on the critical path, could allow a

higher clock frequency. The changes in cache architecture should allow either a large reduction in cache power consumption due to the reduction in area, or a greatly increased cache size with power reductions from decreased traffic on the external memory bus (or some trade-off between the two) – either way the power efficiency should be increased.

5.2.3 Evaluation

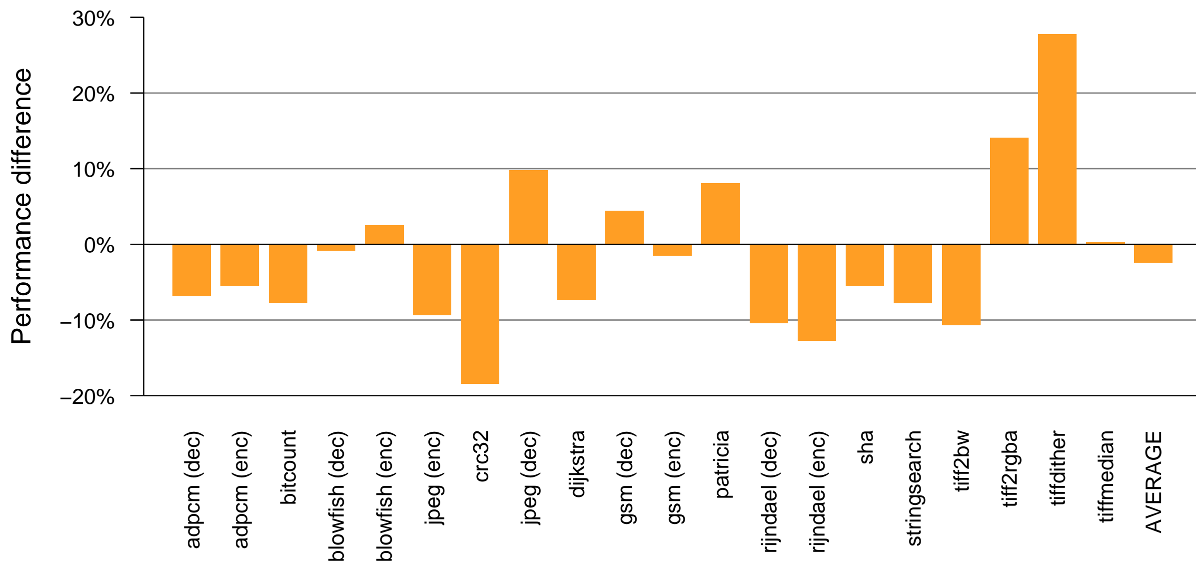
A new machine variant was created in the ILDP toolchain to support the modified ISA; the 'multilib' feature of the GNU binutils made this a simple process, as the standard libraries are automatically built for both ISA variants, and a single compiler switch is used to select the ISA version to use. The simulator was augmented to simulate the architectural changes described in the previous section – again, a command-line switch was implemented to select the appropriate architecture, to make side-by-side comparisons easier.

Figure 5.6 shows the performance of the separated memory version of the architecture, with in-order issue in the MAU. In both the 2-way and 4-way issue width cases the average performance is 2–3% lower than the original ILDP model, but there is a significant variation within the results. In both cases the *tiffdither* test performs significantly (more than 25%) better, and in the 2-way issue case the *jpeg (dec)* test shows a similar improvement. Several tests perform at least 10% worse – in particular the *crc32* test performs 18.4% worse on the 4-way model and 35.5% worse on the 2-way model.

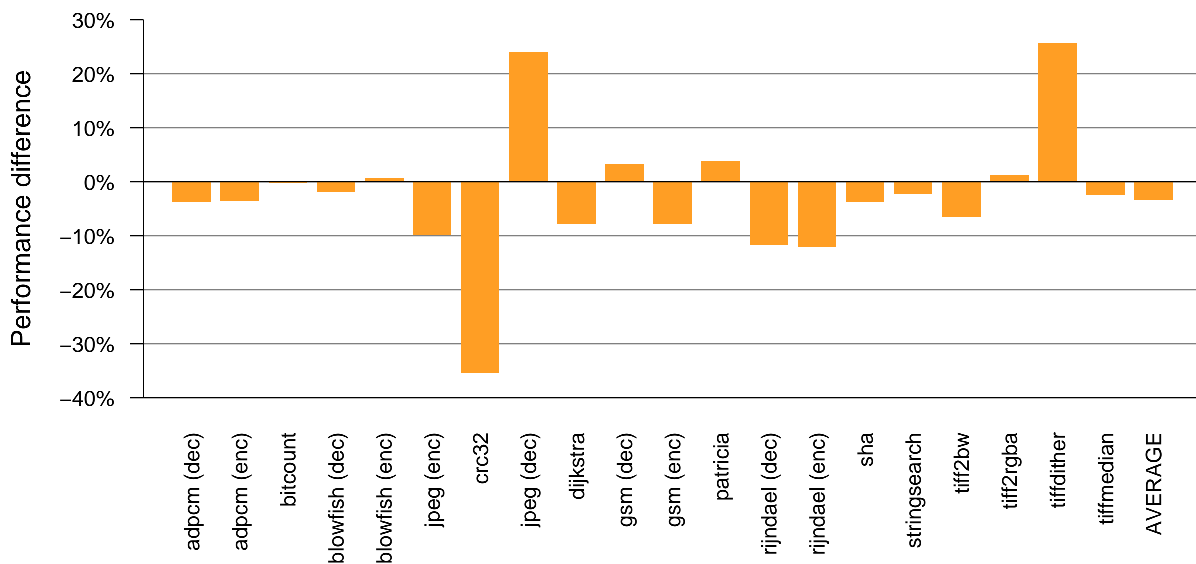
Several possible causes are attributed to the large performance variation:

- The removal of the store queue replication logic holding issue of loads until previous stores had executed – as all memory instructions are executed in order in the MAU, there is no need to stall loads as all earlier stores will have already executed and there is no delay induced by the need to communicate store address values between PEs. Since the MAU only executes memory operations, it is possible that it can execute further ahead in the instruction stream that the PEs and thus possibly resolve these loads sooner.
- The change in the instruction set – since the operands used in memory accesses are now different, some code may compile into longer or shorter code sequences. If this happens within a heavily-executed inner loop in a program, it can have a significant impact on performance.
- All memory operations are now serialised and must execute one at a time in order in the MAU, whereas the standard ILDP model can have an active memory access in each PE – these accesses are not required to execute in order with respect to each other. Thus a load causing a cache miss need not stall processor execution in the original model – the PE executing the load will stall, as will any that then depend on results from the stalled strand. In the in-order MAU model, a cache miss will cause all memory operations to stall, and thus any PEs depending on any data from memory will block.

Figure 5.6 Performance of the MAU architecture variant compared to the baseline



(a) 4-way issue



(b) 2-way issue

In order to evaluate the impact that stalling on cache misses has on the MAU model additional simulations were performed, assuming the implementation of a non-blocking L1 data cache which allows further memory accesses to execute while a miss is being serviced. Tests were run with a memory system that would allow 'hit-under-miss' functionality where a load can be issued while there is a single miss active, and one supporting 'miss-under-miss' where a load can complete if up to three earlier loads have missed.

These results are presented in Figure 5.7. There is no significant difference between the original model and the models which permit non-blocking access to the cache – the *patricia* test shows a performance improvement of approximately 0.9%, while all other tests show a performance difference of less than 0.1%. This is attributed to two main factors:

1) The hit rate on the L1 cache is very high – in many cases greater than 99.5%. With such a low proportion of misses, the performance improvement of non-blocking cache access will be negligible.
2) The memory accesses causing cache misses may well show significant spatial locality – if a benchmark attempts to load an array of data and the first access misses, then the subsequent accesses to nearby memory locations are also likely to miss, thus negating any benefit from being able to continue issuing loads after a miss.

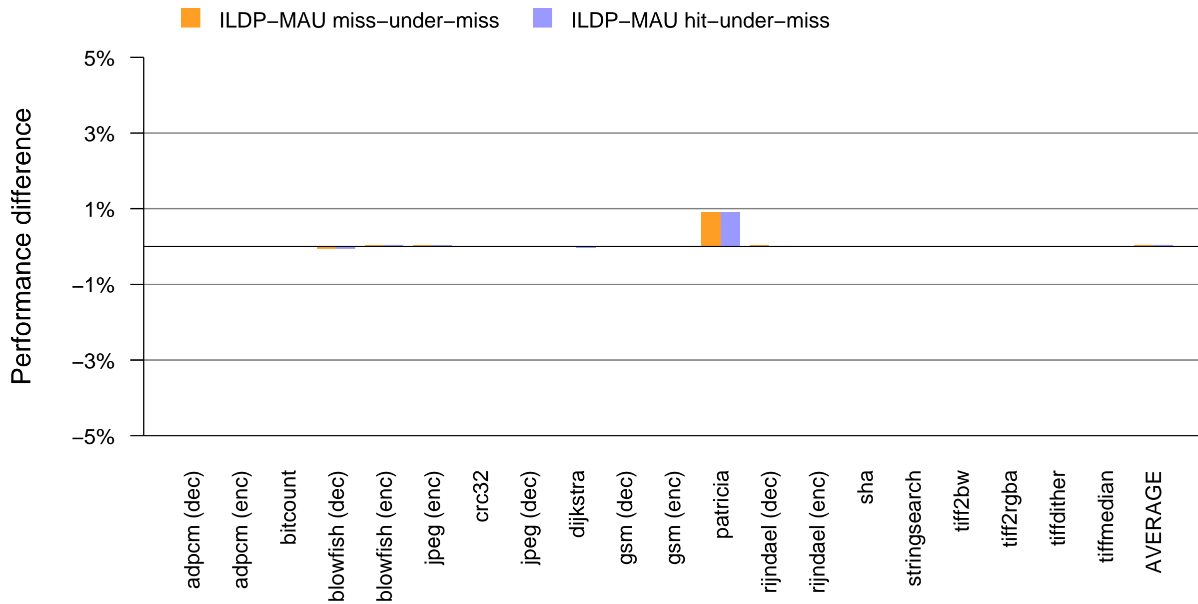
Rename map bandwidth

Further simulations were performed, analysing the effect that the register rename bandwidth has on performance with the MAU model. The simulator was configured for one set of tests to have full rename bandwidth, allowing a full set of instructions to be renamed if they access two registers (this is the configuration used in the previous section). The other set of tests limits the rename bandwidth to be the same as the standard ILDP architecture – a memory operation in the rename pipeline stage will limit the rename bandwidth for that cycle. Figure 5.8 shows the results of these simulations.

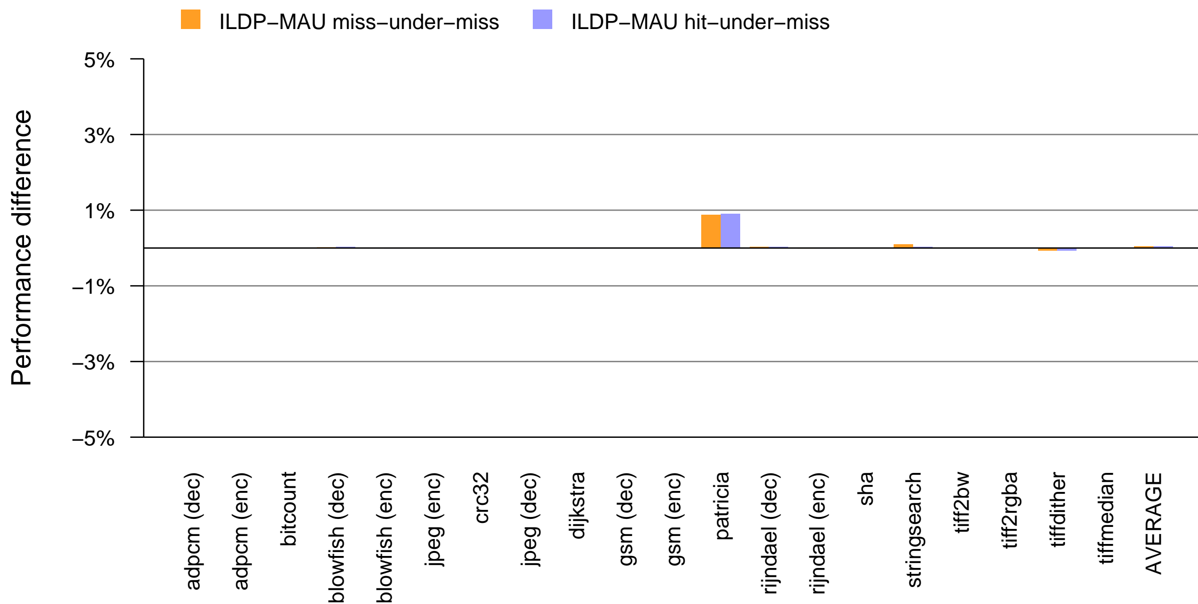
The results show that there is very little performance impact caused by reducing the rename map bandwidth to four ports in the 4-way issue model (less than 1% across all tests) or three ports in the 2-way issue model (less than 1% on average, up to 2.3% for the *patricia* test) – the proportion of memory accesses in the instruction stream is sufficiently low and the number of instructions that do not access the register file is sufficiently high that the vast majority of instructions are not affected by reducing the bandwidth. The 2-way issue model with only two rename map ports does suffer from a noticable performance reduction – on average it performs 4.2% worse, with the worst affected test being *stringsearch* which performs 9.6% worse.

Ultimately these results show that changing the ISA over to the MAU model does not require adding many ports to the rename map, and thus increasing its complexity – virtually identical

Figure 5.7 Performance of the MAU variant when multiple outstanding loads are implemented

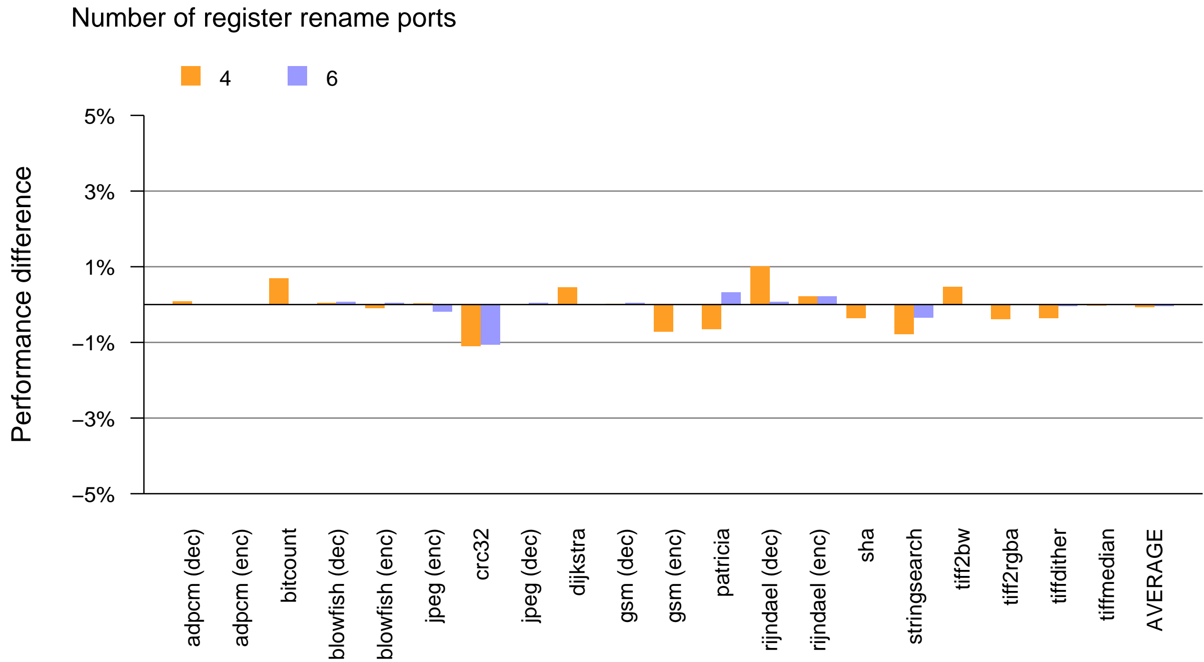


(a) 4-way issue

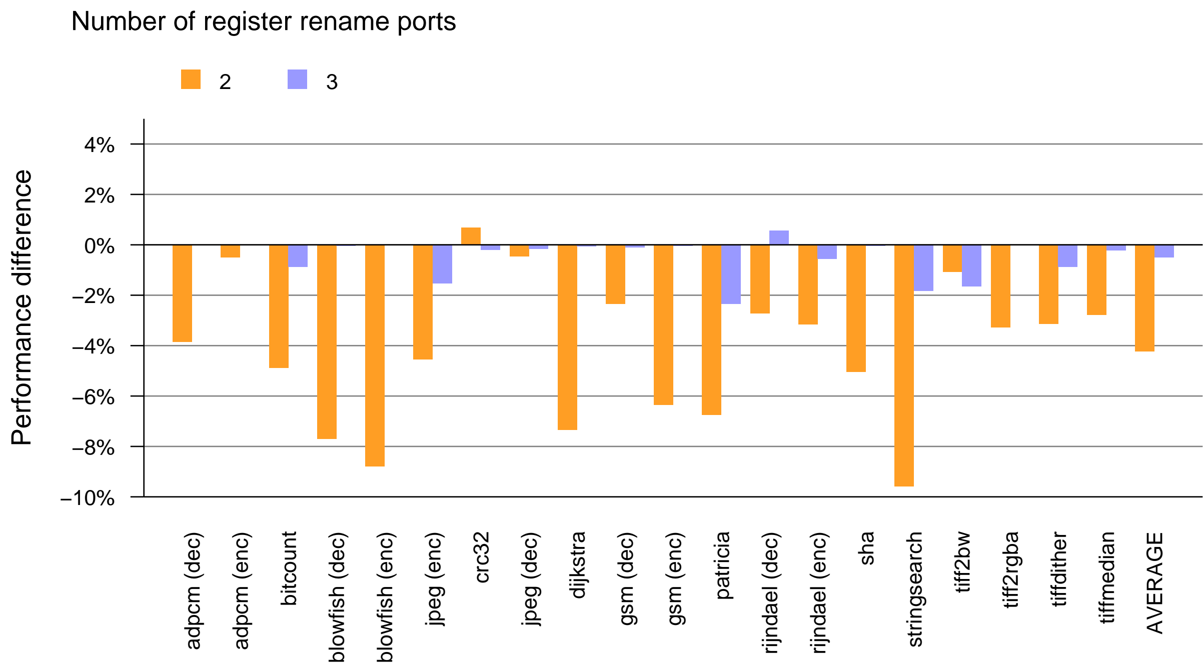


(b) 2-way issue

Figure 5.8 Effect of reducing register rename bandwidth on performance in the MAU model



(a) 4-way issue



(b) 2-way issue

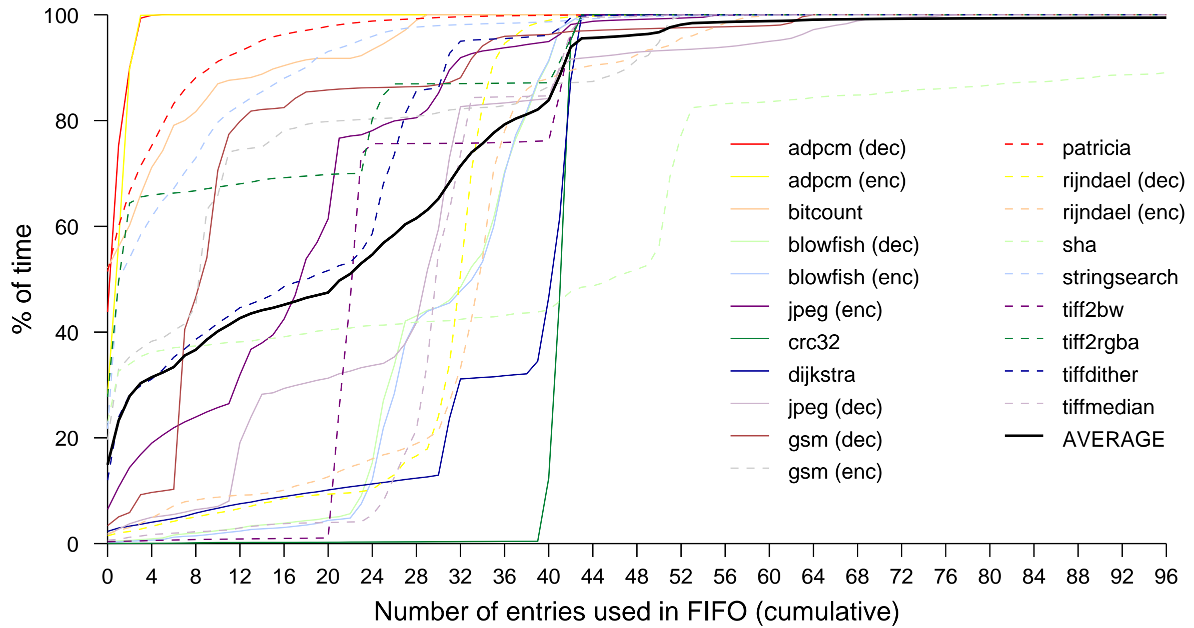
performance is achieved with no additional ports in the 4-way issue case and only a single extra port with 2-way issue.

Instruction FIFO size

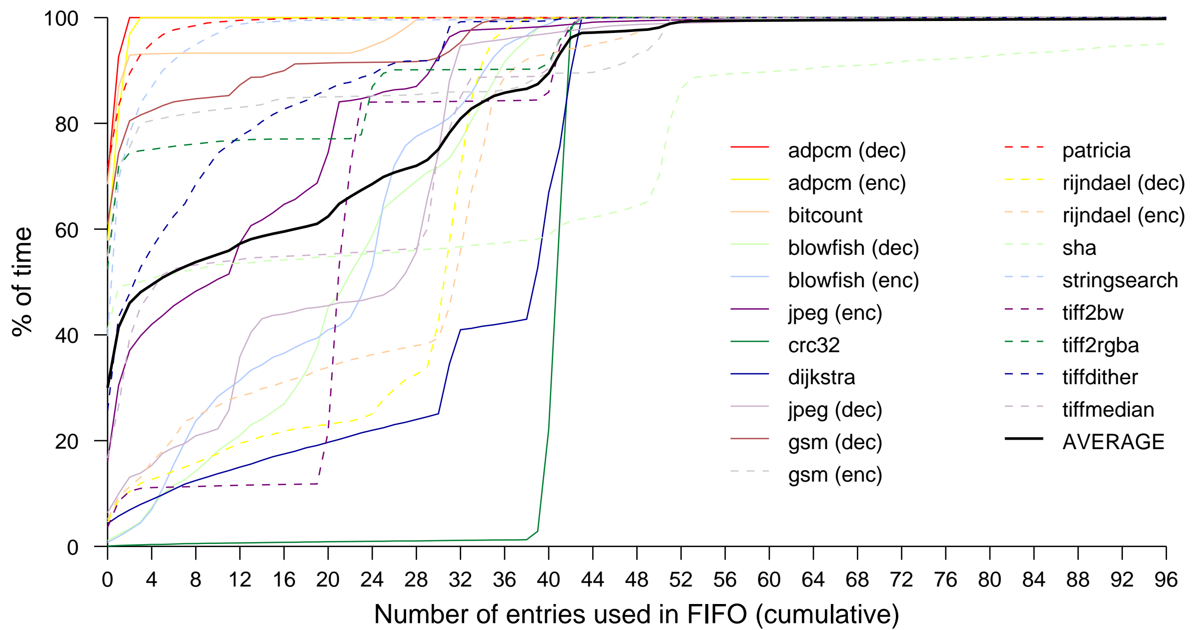
Experiments were also performed to find the optimum FIFO depth for the MAU. Since the ratio of memory instructions to non-memory instructions is not 1:1 and the non-memory instructions are distributed across several queues, the optimum depth for the MAU will likely be different from the other PEs. First the distribution of MAU FIFO occupancy over time was analysed by running simulations where the MAU FIFO depth is unbounded, as shown in Figure 5.9. These results show that on average for 90% of the time in the 4-way issue model there are 42 or fewer instructions in the MAU instruction FIFO, and with the 2-way model there are 41 or fewer instructions in the MAU FIFO 90% of the time. The various tests show a range of behaviours – the *adpcm* tests run using very few FIFO entries (less than four for the vast majority of simulation time), while the *dijkstra* and *crc32* tests show a much higher usage – *crc32* rarely dips below a MAU FIFO level of 40 entries. These results imply that a MAU FIFO depth of 32 entries should be sufficient for most tests, with a smaller size likely to perform well depending on what other factors in the system will limit performance. Section 4.3.3 showed that a PE FIFO depth of 8 entries was sufficient for the PE instruction queues, as they need to be able to hold a complete instruction strand; the MAU FIFO will need to be deep enough to hold all the memory access operations encountered in the instruction stream when reading in a complete instruction strand per PE. Simulations were run fixing the MAU FIFO size at 1, 2, 4, 8, 16 and 32 entries – all tests were run with the PE instruction FIFO size set to 8 entries; the results are presented in Figure 5.10 compared to the unbounded case.

In general there is little performance difference going from an unbounded MAU FIFO down to one with only eight entries - the *blowfish* tests actually show a performance increase with an 8-entry MAU FIFO using 4-way issue, as does the *crc32* test with 2-way issue, but in general the performance is within 1% of the case where the MAU FIFO size is unrestricted. With a 4-entry MAU FIFO the performance is generally similar to the 8-entry case, but a couple of tests start to show slight performance degradation (although the *tiff2bw* test actually performs better). When the MAU FIFO size is reduced below four entries a number of tests show a significant performance reduction, although the *crc32* test stands out once again, as it performs better with a single MAU FIFO entry than with an unbounded MAU FIFO size. Overall these results imply that the optimum size for the MAU instruction FIFO is between four and eight entries, which provides performance very close to the unbounded MAU FIFO model.

Figure 5.9 MAU FIFO occupancy by time

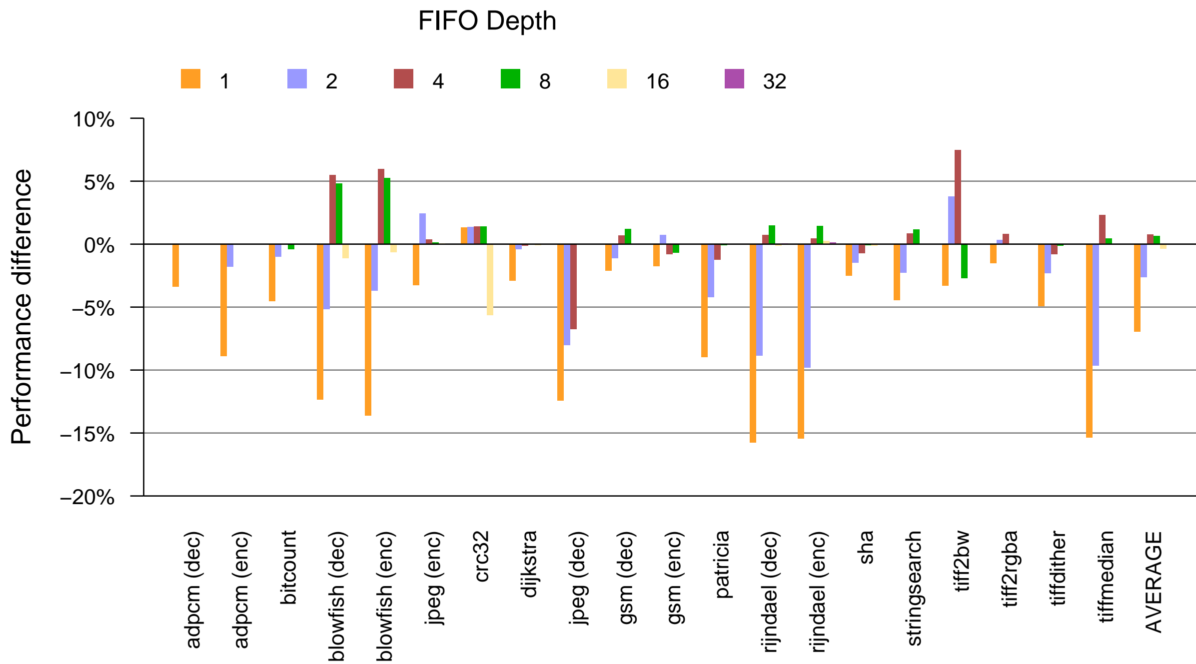


(a) 4-way issue

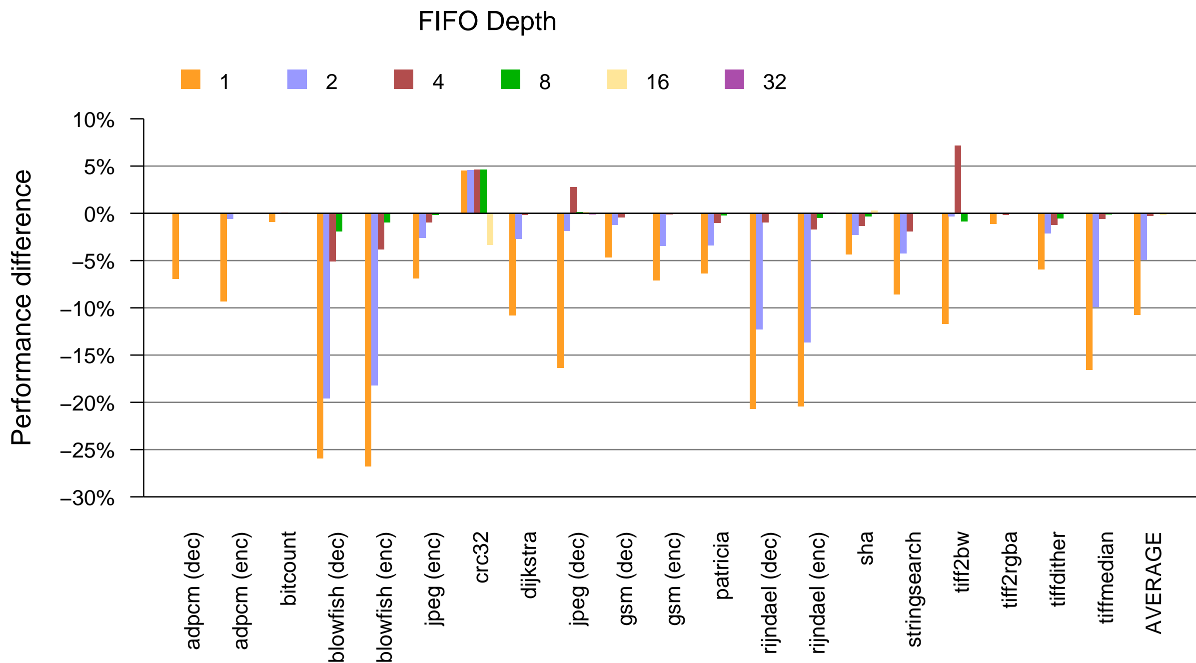


(b) 2-way issue

Figure 5.10 Performance at various MAU FIFO sizes, compared to an unbounded FIFO



(a) 4-way issue



(b) 2-way issue

5.2.4 Further developments

In order to improve performance of memory accesses, which can form a bottleneck for processor performance, a further development on the separated memory architecture is to implement full out-of-order execution for the memory access unit, turning the instruction FIFO into a small instruction window. Instructions execute in two phases – the address computation executes as soon as the source operands for the address are ready, without having to wait for memory resources to be available. Loads can potentially take their results from earlier stores in the window to the same address, although this option has not been investigated here. If the L1 cache permits non-blocking accesses, then load instructions need not be forced to wait by earlier loads that caused a cache miss – loads that miss are flagged as such in the window and are either allocated a miss handling register, or wait for one to become available so they can be reissued. This allows loads that hit in the cache to execute ahead of several earlier accesses that miss.

The MAU approach is analogous to a decoupled architecture [47], where the memory access and general processing instructions are handled by separate processors. Since memory access instructions are more likely to involve long latency operations, they may well benefit more from an out-of-order execution engine, while the other instructions will execute well on the simpler logic of the distributed processing elements.

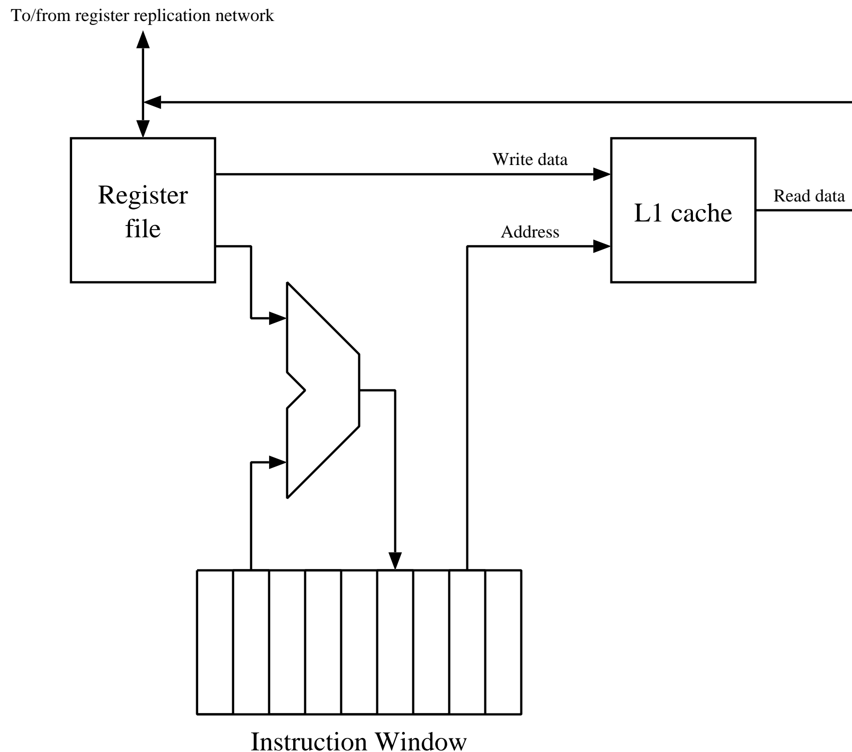
5.2.5 Implementation details

The memory access unit is set up as a small out-of-order issue processor, as shown in Figure 5.11. The front-end pipeline of the main processor issues instructions into an instruction window in place of the instruction FIFO in the earlier MAU design. Instructions execute out-of-order onto three functional units: an ALU for address calculation, a read unit to fetch from the cache and a write unit to place data into the write buffer.

Two types of load/store dependency handling are evaluated. The simpler method is to prevent issue of any loads ahead of earlier stores – while loads can execute out of order with respect to each other, they must wait for stores to resolve in order. In the more complex approach, loads can issue ahead of earlier stores once the address has been resolved and found not to match. In both models the address computation is performed fully out of order.

The address calculation functional unit scans the instruction window for operations with available address operands, then selects the earliest such instruction, performs the address calculation and finally stores the resulting address back into the instruction window. When instructions are placed in the window they are flagged to indicate they require an address computation and to indicate whether the register they depend upon is currently available. When registers become available, instructions depending on them are flagged as being ready for address computation. The selection logic for the ALU generates request signals for all instructions that have both the

Figure 5.11 Design of memory access unit, using out-of-order issue



'address calculation required' and 'register operand value ready' bits set, and puts them through a priority encoder to select the oldest for execution.

The memory read functional unit scans the instruction window for load instructions which have had their address calculated and are not behind any earlier stores (or just stores that may conflict if the more sophisticated dependency scheme is used). The earliest available load is then issued to the L1 cache and will broadcast its result over the register communication network when it is returned.

The memory write functional unit executes stores in order as their operands become available. If the oldest store has both address and data operands ready, it will be issued. The address is checked to ensure that a page fault or other exception will not be raised, and then the store data is written to the write buffer.

If the L1 cache supports non-blocking, or 'hit-under-miss' accesses, then a load that causes a cache miss is retained in the instruction window and flagged as a miss. When the L1 cache has fetched the value and notifies the processor, the load is then reissued and completes. While the cache is servicing the miss, other load instructions can be issued – the issue logic will check that instructions are not flagged as misses in addition to checking the availability of their

address operands. If a subsequent load misses to a different cache line while the L1 cache is still processing the first miss, the load is written back to the window and flagged as a subsequent miss – when the data from the first miss is returned, the first 'subsequent miss' is unflagged, making it eligible to be reissued.

If hoisting loads ahead of stores is implemented, the following additions are made to the dependency logic: when an address is calculated and written back to the window, an associative lookup is performed to locate other instructions accessing the same address (to reduce implementation complexity, only a low-order subset of the address bits are compared – this will identify some false dependencies, but is safe). For a store, this operation is used to resolve loads which might depend on the result of that store. For a load, earlier stores are checked for whether the load may depend on them.

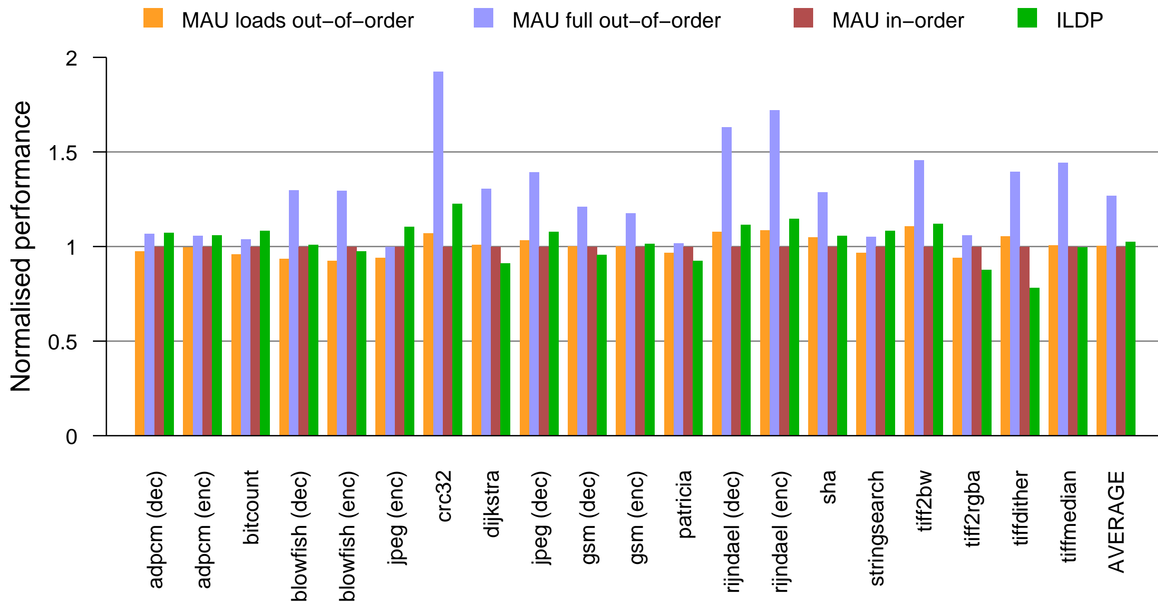
To resolve these dependencies each load is masked by a number of dependency bits. When a store is placed in the window, it is allocated one of these bits and loads issued after the store have this bit set to indicate a possible dependency on that store. When a store address is computed, the address is compared against the addresses of later loads and any that differ have their dependency bit corresponding to the store cleared as they cannot depend on that store. Similarly, when a load address is computed, the address is compared to earlier stores – for any stores with a different address, the corresponding bits are cleared in the load's dependency mask. When a store retires, all loads have the bit corresponding to that load cleared (another option is that the bits are cleared when the store executes, if loads can read data from the write buffer). A load is only eligible for issue when it has a clear dependency mask.

5.2.6 Evaluation

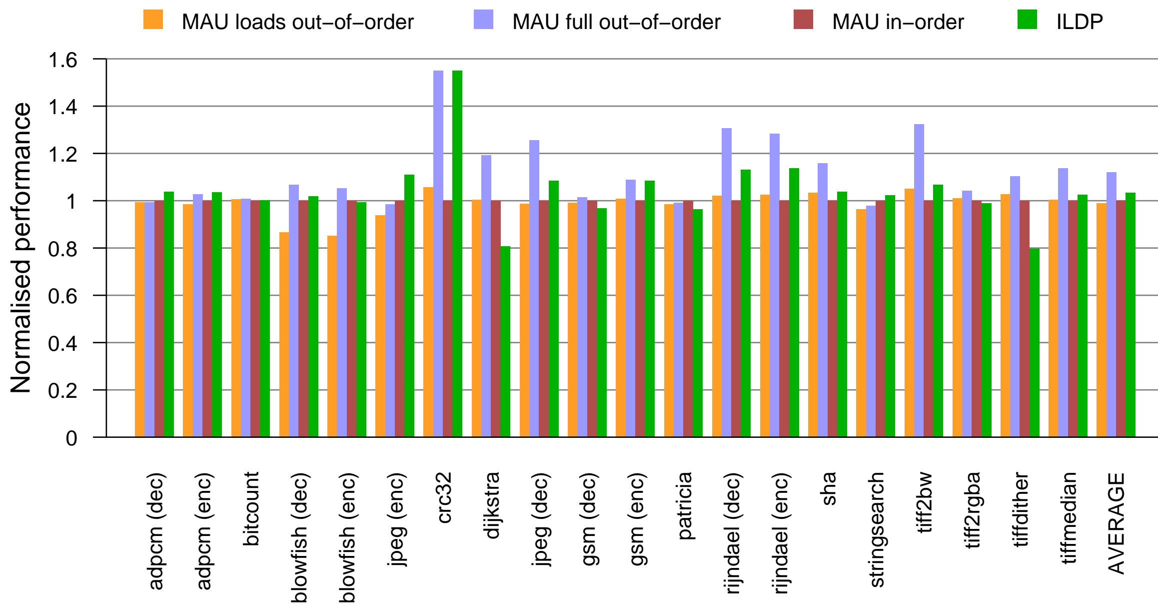
The implementation in the previous section was added to the simulator and the performance compared to the baseline and MAU architectures. The performance was evaluated both with and without non-blocking access to the cache and with both models of load/store dependency resolution (whether or not loads can overtake independent stores). Figure 5.12 shows the results of these tests.

These results show that the out-of-order MAU model provides comparable performance to the in-order model when loads are not allowed to issue ahead of stores (in some cases the performance is worse – this is likely due to subtle differences in the implementations within the simulator model). When loads are permitted to overtake independent stores, several tests show a marked improvement in performance (up to 72% faster than the in-order MAU). In particular, those tests that performed worse under the in-order MAU model than in the baseline ILDP model regain or exceed the original baseline performance – the out-of-order MAU model averages 23.9% faster than the original ILDP model. This would indicate that the critical factor in the memory system performance is resolving dependencies between loads and stores, in order to allow loads to issue as early as possible.

Figure 5.12 Performance of various models of out-of-order memory access unit



(a) 4-way issue



(b) 2-way issue

The out-of-order MAU approach will necessarily have higher complexity than implementing an in-order memory PE. The smaller size of the instruction window for the memory access unit as compared to that for a full superscalar processor means that the resulting complexity will not be as great as using a superscalar implementation for the whole processor. Contrasting the complexity of the out-of-order MAU design with the baseline ILDP architecture is more difficult – the added complexity of the issue logic for the MAU is offset somewhat by the removal of the store queue replication network and the replicated L1 caches and their association coherency logic. This approach should either use less die area, or use the same die area more efficiently – even with the more complex memory issue logic, the need for replication of caches or implementation of multiported caches is removed.

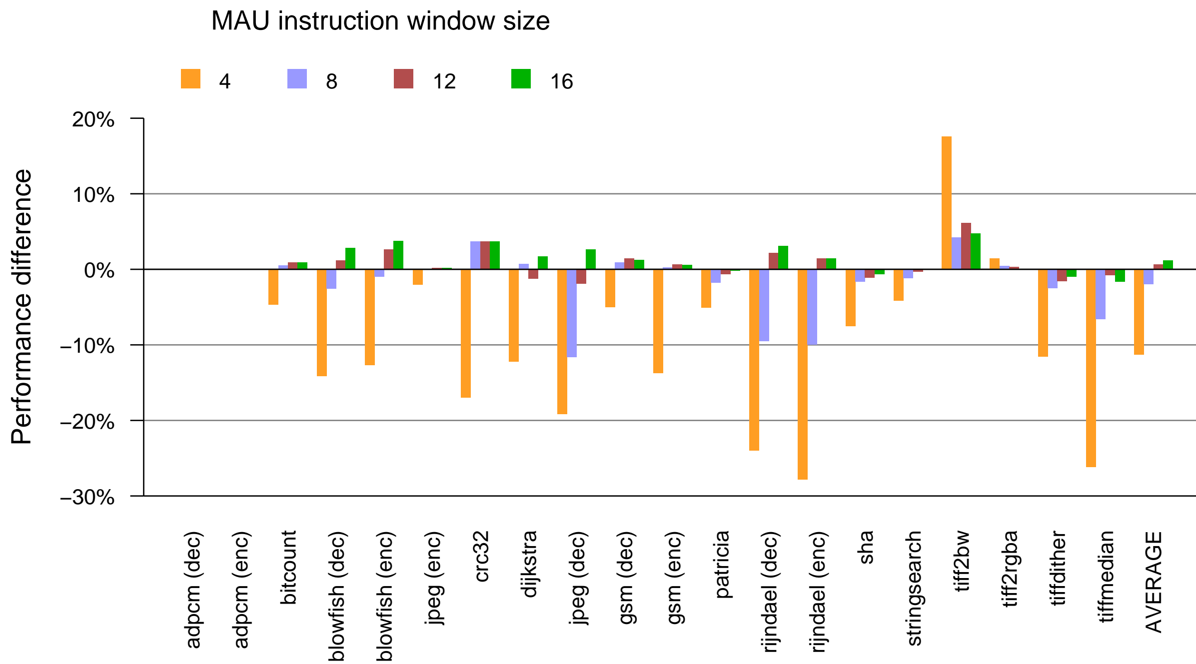
Instruction window size

Since the complexity of the issue window logic scales quadratically [42] in the size of the window, it is important to keep this window size small. Section 5.2.3 indicates that a depth of between four and eight entries for the instruction FIFO is sufficient with the in-order MAU. Experiments were conducted to investigate the effect of window size on performance, the results of which are presented in Figure 5.13.

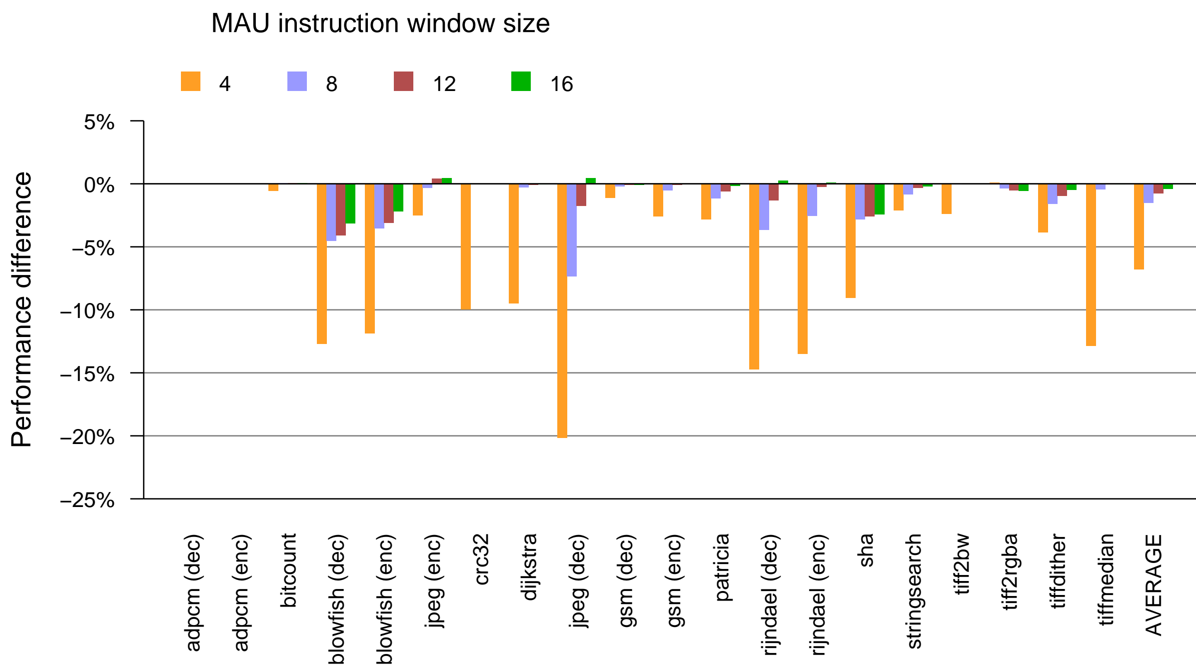
These results show that, on average for the 2-way model, a 16-entry instruction window provides 99.6% of the performance of an unbounded window, while a 12-entry window achieves 99.2%. Scaling back to eight entries reduces performance by 1.5%, and a 4-entry window performs 7.8% worse than one with no limit; the results in the 4-way case are similar. These results imply that a window size of twelve entries is ample to achieve nearly optimal performance, while a reduction to eight entries still provides good performance with a large potential complexity reduction. The *tiff2bw* test actually runs 17.6% faster with a 4-entry window – this result is somewhat unexpected, and further examination has been unable to explain the reason for this apparent discrepancy. Since this is the only data point to display this behaviour, it is considered to be an outlier and has been discounted.

If the instruction window size proves to be a critical point when it comes to trading off between complexity and performance, another possible approach is to use a smaller instruction window fed by a simple instruction FIFO. This will have lower performance than a large instruction window as fewer instructions can be reordered (although the relative sizes can potentially be optimised so that this performance drop is minimal) but will have much lower complexity than a full window. The performance will be greater than a single small instruction window as the issue logic does not need to stall when the window becomes full – the instruction can be placed into the FIFO, and will be moved into the window when space becomes available. Based on the results presented in this section, if this approach were to be implemented, an instruction FIFO of 4-8 entries feeding a 4-entry window seems like a suitable trade-off.

Figure 5.13 Effect of out-of-order MAU instruction window size on performance



(a) 4-way issue



(b) 2-way issue

5.2.7 Cache structure

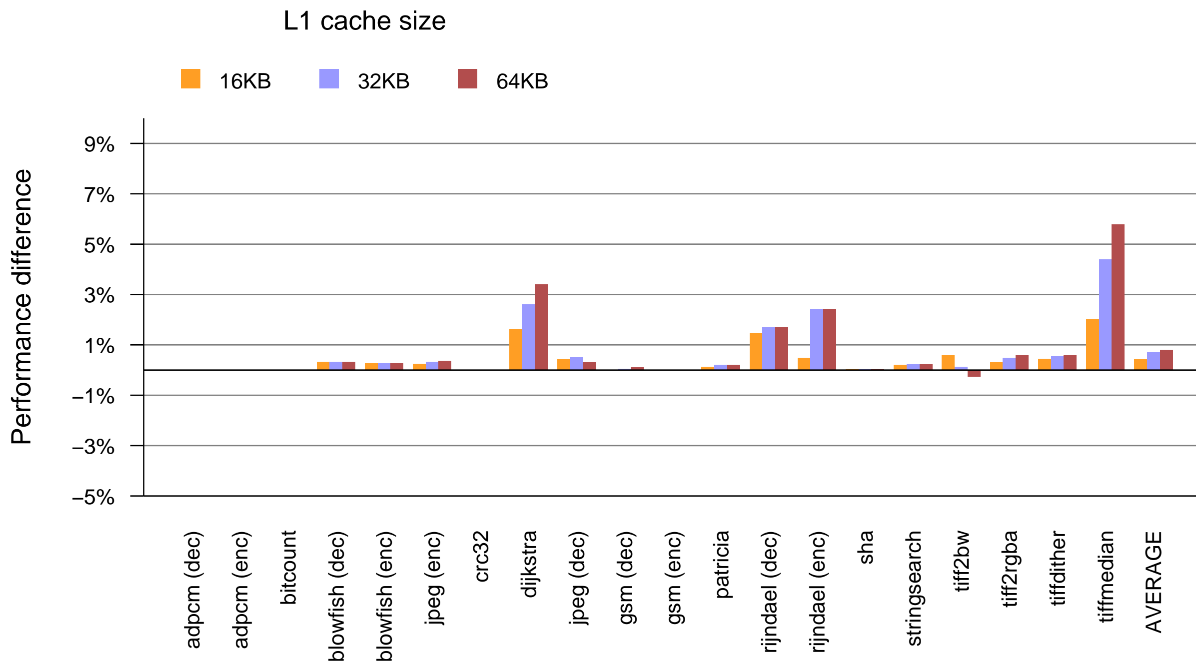
In section 4.3.7 the effect of varying the associativity of the L1 cache is analysed. Scaling the design from multiple replicated caches to a single cache makes it more feasible to increase the cache size. This section investigates the effect cache size has on performance in the MAU design. The original ILDP design contains a total of 64KB of L1 cache, partitioned into eight blocks of 8KB each. Experiments were conducted with a single L1 cache of 64KB total size, as well as a smaller cache of 32KB or 16KB. The performance of each of these configurations was then compared to the results with a single 8KB cache. All tests were run using the out-of-order MAU model, with an 8-entry instruction window.

The results from these tests are presented in Figure 5.14. As in section 4.3.7, there is only a small variation in performance. When using 2-way issue, the *dijkstra* and *rijndael (dec)* tests perform slightly over 1% better with the larger cache sizes. The 4-way issue case seems more bound by the cache parameters – in particular the *tiffmedian* test (which was the only test to show a noticable performance improvement when the associativity was increased) is 5.8% faster when run with a 64KB cache. Overall, the performance improvements are marginal, which is again attributed to the low memory footprint of the MiBench suite.

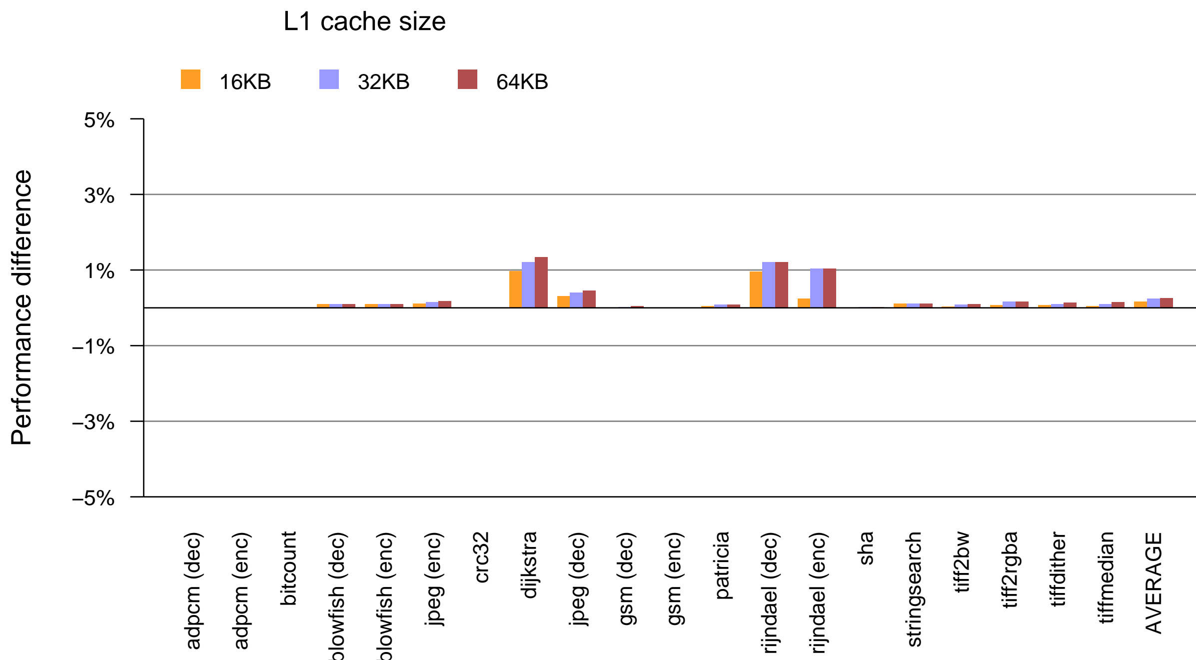
5.2.8 Conclusions

The approach of separating memory accesses out into a different processing unit is explored in detail using several approaches. A simple implementation issuing memory operations in order can result in a large reduction in silicon area usage and provide a very similar overall level of performance. More sophisticated schemes can be used that allow reordering of load instructions to prevent stalls on register values or cache misses. At the cost of higher complexity, these more advanced designs provide greater performance (up to a 24% improvement over the base model using 4-way issue) while still having a greatly reduced area.

Figure 5.14 Effect of increasing L1 cache size on performance (compared to 8KB size)



(a) 4-way issue



(b) 2-way issue

5.3 Register inlining

The base ILDP architecture replicates the register file, having a copy in each of the processing elements. With 128 physical registers and 8 PEs this means there are a total of 1024 registers on the processor. Even if the size of the physical register file is scaled back to 32 registers, the overall processor will require a total of 256 registers. Since all register values are replicated to all PEs, but most values will be used by only one or two PEs, most of the values in the register files will be effectively dead – this amounts to a huge waste of both area and power, both in terms of static consumption from holding these useless values and dynamic consumption from writing them into the register files. Even if power gating is utilized on inactive PEs, the register file and replication logic must still be powered and kept up-to-date, in order that register file contents are valid when PEs are awakened.

Lipasti et al. [36] describe a scheme for improving the performance of superscalar architectures whereby registers containing small values can be stored directly in the reorder buffer in place of the register number. This bypasses the register file access when the instruction is executed, reducing the latency of instructions using registers containing small values (which form the majority of data values held in registers). While their work is targeted on improving performance, this approach can be used to reduce complexity, as it potentially allows the register file to have fewer ports, or allows a multi-cycle latency register file to be implemented using less aggressive logic. While such a design would reduce the register file performance, this would be mitigated by the reduced access frequency. The lower complexity could also possibly reduce overall cycle time, improving overall performance.

Using a similar technique, this section proposes a development to the ILDP architecture to reduce the overall silicon area usage. The register file in each PE is eliminated and instead the instruction FIFO is augmented with an associative memory for holding the register operand tags. Rather than using inlining to reduce register file latency, here it is used to reduce storage replication and thus save die area and reduce power consumption. When a register value is received by the PE, rather than being written into the register file, an associative lookup is performed on the instruction FIFO to find any instructions depending on that value. For those entries that match, the data value is then written into the FIFO alongside the matching instructions. In this scheme, only live data would actually be stored within each processing element – while some values may be replicated within the FIFO, energy is not spent storing data values which ultimately are not used. As the FIFO size (e.g. 8 entries) is much smaller than the overall register file size (32–128 entries), this should greatly reduce overall silicon area.

A single copy of the full register file is maintained – this contains the only copies of old registers that no longer hold the values of any architectural registers at the issue point, and thus are speculatively dead, but may actually be needed in the case of an exception causing rollback to a point where they are in scope. This register file is accessed by the issue logic after register renaming and before the instruction is steered. Instructions depending on data values already available in the register file read them at that point in the pipeline, while those requiring data

not yet computed are issued to the processing elements along with the tag of the register they depend upon, and are updated later in the instruction FIFO when the value becomes available.

This system should allow for large power savings, as any idle processing elements can be entirely powered down, without needing to keep a register file and update logic active. Even active PEs will consume less static power, as the memory for storing operand values inline in the FIFO is smaller than a copy of the register file.

5.3.1 Implementation details

In this scheme, each processing element is modified by removing the register file and replacing it with an enhanced version of the instruction FIFO. In addition to the standard FIFO control logic, the augmented instruction FIFO contains several blocks of RAM and some logic for performing the associative lookup. The FIFO must store, for each instruction:

- A tag identifying the individual instruction
- The opcode of the instruction
- If the instruction writes to a general-purpose register, a tag identifying that register
- Any immediate operand value it may have
- For instructions accessing a register operand, either the tag identifying the register providing the operand or the value of the register operand itself
- A flag to indicate whether the instruction is waiting on a register value

The tag, opcode and immediate values are stored as in a standard FIFO: in a RAM with one write and one read port and some logic to generate the address pointers for these ports and to handle empty/full notification. The register values and register tags are stored in separate RAM arrays, sharing the read and write ports of the main FIFO, with an additional write port for register value updates from the inter-PE communication network.

When a register value is received, the register tag is asserted onto the associative lookup port of the register tag array. Comparators in each row compare the incoming tag with the stored tag, and all entries that match drive an associated wordline. These wordlines drive the additional write port on the register value array, writing the new register value into each location of the instruction FIFO that depends on the register being updated. A one-bit array of flags is maintained to indicate whether a row is waiting on a register value – this bit is cleared when the register value arrives, and an instruction will only issue if it is not waiting on data.

Figure 5.15 Register tag/value portion of PE instruction FIFO

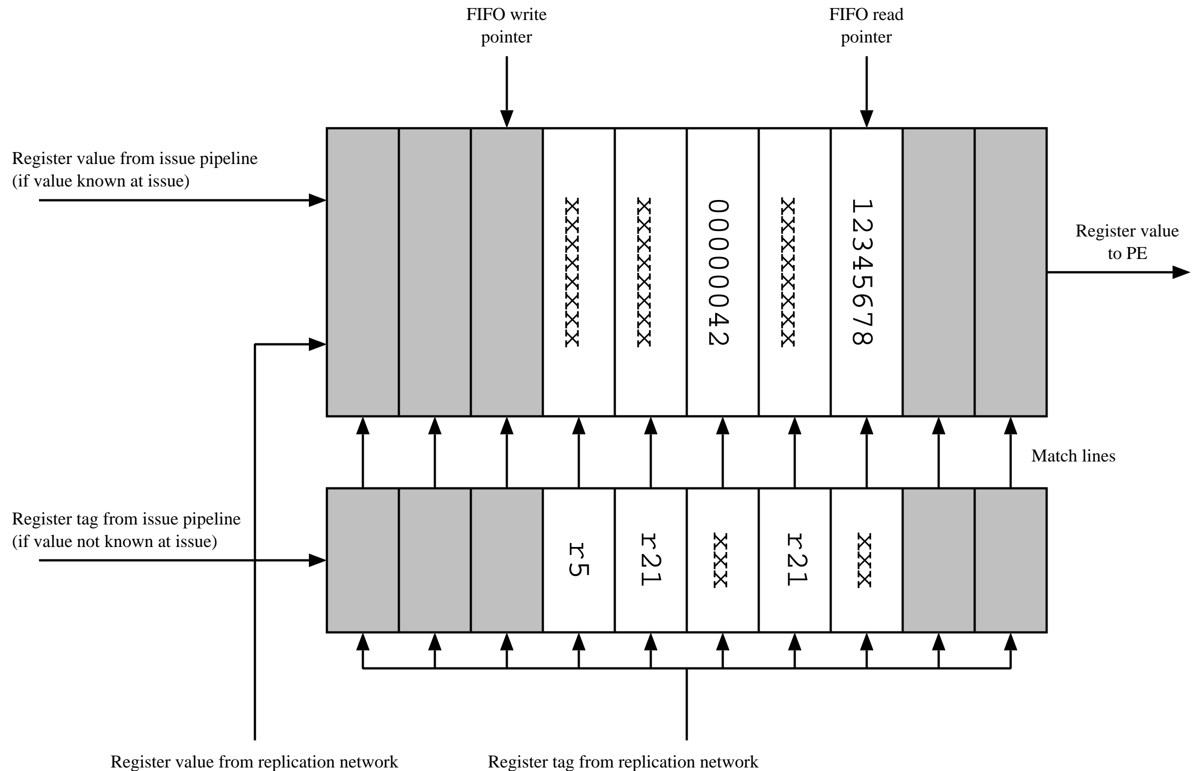


Figure 5.15 illustrates the organisation of the register value and tag portions of the instruction FIFO. In the figure there are five instructions currently stored in the FIFO. The instruction at the head of the FIFO has already had its register value resolved, and is thus available for issue; the third instruction also has its source operand present in the FIFO. The second, fourth and fifth instructions are all still waiting on data to become available. If, in the next cycle, the value of r21 becomes available, then the tag for r21 will be asserted to the associative tag RAM. This will match the two entries containing the r21 tag, and thus drive the associated match lines high. The value of r21 will be asserted to the write port for the value RAM, and thus the value entries corresponding to the second and fourth instructions in the FIFO will be updated, while the entries will be marked to indicate that the values have been resolved and thus the instructions can execute.

A small lookaside buffer is maintained in each PE to store any values received in the last cycle or two (depending on the delay between the register lookup in the issue pipeline and an instruction being written into the PE's instruction FIFO), so that instructions that are 'in flight' between the register access stage in the issue pipeline and the tail of the instruction FIFOs do not miss register updates. When an instruction is written to the tail of a FIFO, this buffer is accessed to see if it holds the value of any source registers of the instruction.

While in the baseline architecture there are copies of the register file in each PE, in this approach there is a single register file, accessed in the issue pipeline (probably in the 'steer' stage, but depending on timing an additional stage might need to be inserted between 'rename' and 'steer'). If an instruction being steered reads a register value that has already been computed, that value is read from the register file at this point and the instruction is placed into the destination FIFO having already had its input dependencies resolved. This register file is updated as it receives the data broadcast over the register communication network. Thus the file requires one read port for each instruction that can be in the steering stage at once (although theoretically fewer ports could be implemented; see Section 4.2.3) and one write port per value that can be sent over the register network in a single cycle.

5.3.2 Complexity

The logic required to implement the associative FIFOs is analogous to the associative logic on an instruction window in a superscalar processor to tag ready instructions, but with only a single associative write port. The reduced size of the individual instruction FIFOs compared to a superscalar instruction window (the size of all the FIFOs added together is comparable to the size of a full instruction window) should reduce the delay of this structure – the delay of such an associative memory is quadratic in the number of words, although for small issue widths the quadratic term is not very significant and the scaling is closer to linear [42].

The main delays through this structure are the bitline delays on the read and write ports, the wordline delays and the delay through decoder or comparator logic. For the non-associative accesses on a FIFO pop, the delay consists first of the delay through the address decoder, then through the wordline from the decoder, and finally through the bitlines to the output connections. For a FIFO push, the bitline drive time can occur concurrently with the decoder and wordline delays. With an associative write, first the bitlines into the tag memory must be driven, then the comparators must drive the appropriate wordlines. The bitlines into the register value memory can also be driven concurrently with the other delays.

The main register file accessed in the issue pipeline needs to have as many read ports as the register rename map, plus as many write ports as are needed by the register replication logic. Given the number of ports required, this register file could become a problem meeting timing requirements, although the number of ports required is still far less than for a standard superscalar processor of the same issue width.

5.3.3 Evaluation

At the architectural level, this technique should provide the same per-clock performance as the baseline ILDP architecture – this design simply rearranges the storage mechanism for GPRs

and does not introduce any additional delays at the clock cycle level, unless the added register access in the issue pipeline requires adding an additional pipeline stage. At the circuit level, the new FIFO arrangement is more complex than the previous multiple register file design, but the additional complexity is mostly from extra logic; the bitline delays of the instruction FIFOs will be smaller than those of the original register files as there are fewer words in the memory. By reducing wire delays at the expense of greater logic delays, this design should scale better at smaller feature sizes than a register file (as logic delays scale better than wire delays), thus the overall cycle time is unlikely to be significantly affected by this change, as the associative memory should not form part of the critical path.

There should be a significant reduction in power consumption using this technique, as the 128-entry register file in each PE is replaced by a single register file in the issue logic and a smaller associative file alongside the existing instruction FIFO – with a FIFO depth of eight, the associative register memory will contain only eight entries, rather than the original 128. Greater benefits can be realised if clock gating or power gating are also used – since only live data is stored within the PEs then, when an inactive PE is powered down, the register memory and the register file update logic can also be powered down, giving a large power saving if PEs are shut down for much of the time.

5.3.4 Conclusions

Inlining the register file into the instruction FIFOs reduces overall silicon area, decreases static power consumption and should not significantly affect performance, although the performance effect is hard to judge without performing a full circuit synthesis. Rather than being replicated across all the PEs in case it will be required, data is only stored within a PE if it is definitely going to be used in the future. If a PE has an empty instruction FIFO, it contains no live data and thus register file inlining enables further power savings by allowing all of the PEs to be powered down if power gating is used.

Given that using register file inlining allows a PE to be completely powered down, this technique could be used in combination with the FIFO multiplexing from Section 5.1 to allow a highly scalable power management system to be implemented. If some of the PEs are augmented to be able to have more than one strand currently assigned to them, then it is possible to power down individual PEs when the processor is in a low power mode, allowing the remaining powered PEs to handle execution. For example, with four PEs capable of executing two strands and four capable of only one, the core could be configured to run with anywhere between four and eight PEs activated at one time, trading off power consumption versus performance.

5.4 Summary

Several developments on the ILDP architecture are described to reduce the overall complexity of the architecture. Reducing the number of processing elements and assigning multiple instruction FIFOs to them can greatly reduce the total silicon area of the design (the area used by execution resources and L1 cache would be halved), but has a significant performance penalty of 12–14%.

Adding an additional processing element to handle memory accesses and removing the memory logic from the original processing elements greatly reduces area by removing the need for the store queue replication logic and maintaining only one copy of the L1 cache, providing similar peformance with simple in-order issue logic, or a 24% speedup with more sophisticated issue logic.

Eliminating the replicated register file in each processing element and replacing them with a single register file and an associative operand memory in the instruction FIFO allows the total silicon area to be reduced and also allows potentially very good power savings, with hopefully a negligible performance impact.

6 Conclusions

This thesis has evaluated the performance and complexity of the ILDP architecture in the context of an embedded environment. Numerous performance simulations have been run, evaluating the effect of a range of architectural parameters and analysing the suitability of various developments upon the architecture. This chapter summarizes these results and presents several avenues for potential future research based on this work.

6.1 Summary

In Chapter 3 I gave a detailed overview of the ILDP microarchitecture and instruction set, and I provided a qualitative analysis of the complexity of an ILDP implementation, as compared to both a typical scalar RISC CPU and a superscalar design. The ILDP architecture eliminates or greatly reduces the size of a number of structures present in a superscalar architecture, allowing its complexity to compare favourably with a simple scalar design.

In Chapter 4 I compared the performance of an implementation of the ILDP architecture to some typical processors used in high-performance embedded systems. The results show potential, but highlight the need for a better compiler implementation than the one used for these tests. I went on to analyse the effect on performance of a number of the architectural parameters; I have shown that many of the parallel structures used within the processor can be scaled to provide less parallelism with little cost to the overall performance, for example the register renaming stage can be scaled from being able to process four instructions per cycle to three with a drop in performance of less than 0.2%. In particular, the size of the register file can be greatly reduced with little average effect on performance.

I presented several possible developments to the ILDP architecture in Chapter 5. A scheme to halve the number of processing elements, and thus greatly reduce silicon area and power consumption, was outlined but proved to have a significant effect on performance. I presented a method to reduce the area and power requirements of the memory logic in the architecture by adding a new processing element with sole responsibility for handling memory accesses. If this new unit is configured to execute instructions in order, then similar performance to the original architecture is achieved with a large reduction in area and power requirements. If the new unit is set up to execute memory operations out of order then, at an increased area/power cost, a

significant improvement in performance is possible. Finally, a new organisation for the register file was proposed, which greatly reduces the overall area used and has the potential to reduce power consumption, especially in the presence of clock or power gating.

6.2 Conclusion

It has been shown that the ILDP architecture is able to achieve a high degree of instruction-level parallelism – the IPC attained in benchmarks was more than double that of the other architectures modelled. The overall performance compared to these models is less impressive – a 53% improvement over the PowerPC model and a 19% improvement over the ARM model. This is due to the ILDP code for the benchmarks requiring many more instructions than the other architectures, which is attributed in part to the basic features of the ISA, in part to specific choices made in this implementation which could be altered, and in part to the prototype nature of the compiler backend, being unable to optimise as efficiently as the backends for ARM and PowerPC. While it was not possible to simulate against an out-of-order architecture, Kim and Smith [30] show that the baseline ILDP performance is comparable to an out-of-order processor. Overall it is shown that ILDP has the potential for high performance, without much of the overhead present in traditional out-of-order superscalar architectures.

The initial ILDP model was based closely on that presented by Kim and Smith [30], which has fairly hefty resource requirements, which will come at a cost in silicon area and power consumption. I have shown that many of these structures – in particular the general-purpose register file – can be reduced in size (with associated power and area benefits) without unduly impacting upon performance: reducing the register file from 128 to 32 GPRs only results in a 4% drop in performance whilst reducing the area used by the register file by 75%. If the bandwidth of the issue pipeline is then reduced from four instructions per cycle to two, performance falls by a further 8.1%, but the area used by the issue pipeline will be approximately halved, and the reduction in complexity in the issue logic could well allow the cycle time to be decreased, allowing the design to be clocked faster.

I have proposed several advancements to the architecture that can improve performance, reduce resource requirements and in particular allow the design to be very scalable, allowing the trade-off between performance and area/power to be chosen both at design time and, with appropriate power gating techniques, dynamically while executing. Moving the memory access logic from the individual PEs to a specialized memory access unit eliminates any need for replicating copies of the L1 cache (saving area and power) and, if using more sophisticated scheduling logic than the main processing elements, can improve performance by 23.9%. Removing the replicated register files from the processing elements and instead using a more dataflow-oriented technique akin to Tomasulo's reservation stations [58] allows individual processing elements to be powered down without loss of useful state – this enables fine-grained power/performance scaling by controlling the number of active PEs.

It seems unlikely that this particular architecture will be used commercially in the near future – in the present high-performance embedded processor market code compatibility is still a large issue and moving to a new, incompatible, ISA is difficult. However, there is currently a trend towards the use of intermediate bytecode languages such as Java and .NET which are either interpreted or converted to native code using just-in-time compilation, so a switch to an ISA using some of these ideas may well be more likely in a few years. This could provide a number of performance advantages – currently software designed to be portable between various implementations and revisions of an architecture must usually be compiled for the lowest common denominator (x86 binaries are often compiled for the old Pentium architecture in order to ensure compatibility with a wide range of processors), but the compiler is thus unable to use any of the features of newer architectures, or to perform core-specific scheduling. If just-in-time compilation is used then optimizations can be performed for the specific model of processor being targeted.

Ultimately, it has been shown that there is definite merit in an instruction set which allows the compiler to more obviously indicate instruction data dependencies to the processor, grouping dependent instructions into strands which can execute in close proximity. This type of instruction set allows simpler issue logic for out-of-order execution using information provided by the compiler, without the need for the detailed microarchitectural knowledge required when the compiler must perform static scheduling, as in the IA64 architecture.

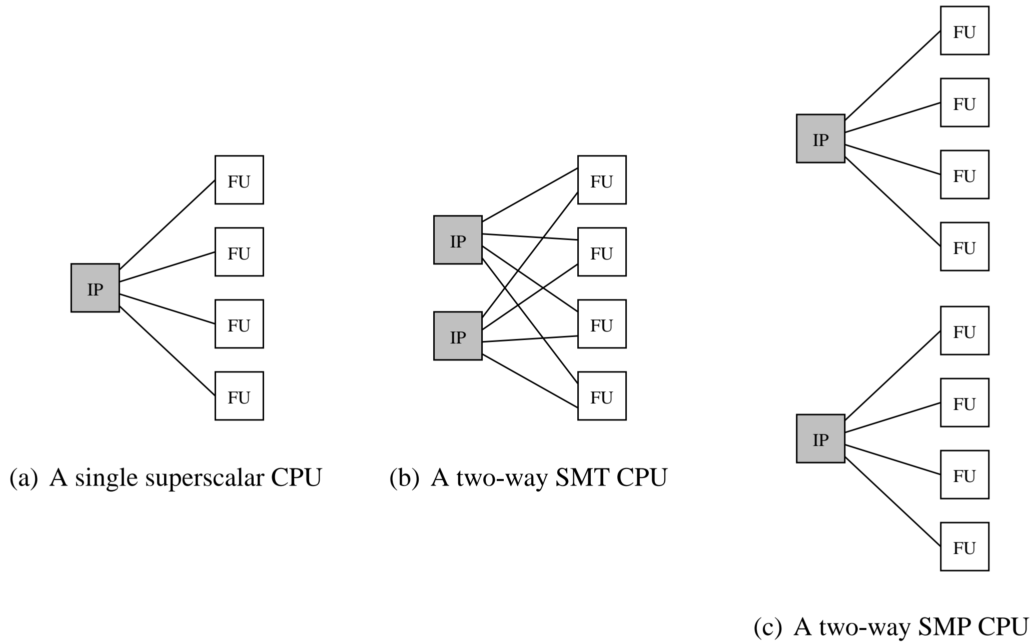
6.3 Future work

There are a number of areas in which future research could continue upon the work presented within this dissertation. In particular, while detailed analysis has been performed on processor performance in terms of number of cycles executed, there is scope for more investigation into the impact various architectural features have upon cycle time. There would probably be merit in synthesizing a design of the ILDP architecture in order to identify where the critical timing paths lie.

Another area which could be further developed is that of the chosen instruction set architecture. It would be useful to perform an analysis showing which features of the instruction set could affect performance, particularly when comparing the performance of the ILDP architecture to that of other architectures. Several of the features of the instruction set proved to cause difficulties when adapting the GCC compiler to generate code for ILDP – some of these features could possibly be changed to allow improved compiler efficiency.

There is considerable scope for investigating how to perform compilation for the accumulator-based ILDP architecture. This research was based on a GCC backend targeted to ILDP, with appropriate fixups and some optimizations in order to support register allocation for the hierarchical register file. A strategy that might be able to generate better code out of the compiler

Figure 6.1 Traditional CPU arrangements for multiprocessing



(a) A single superscalar CPU (b) A two-way SMT CPU (c) A two-way SMP CPU

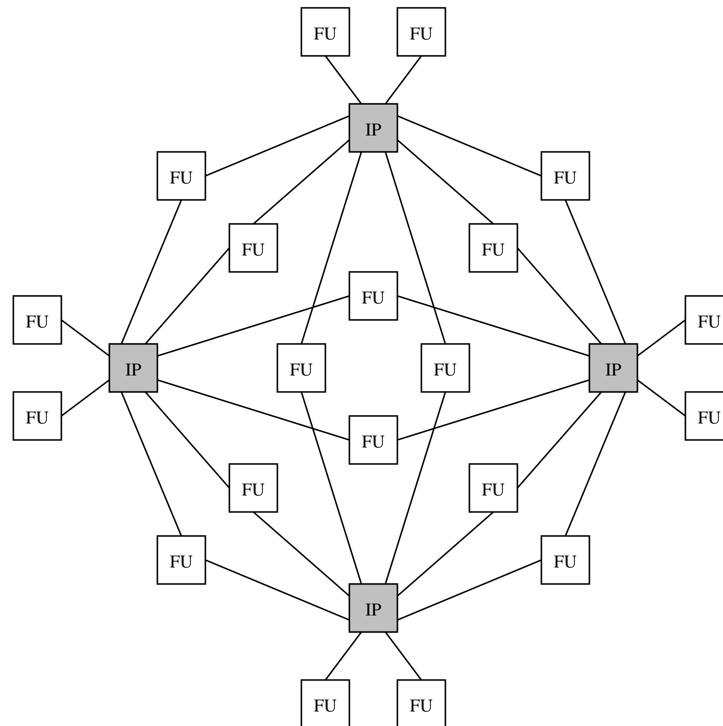
might utilize an additional compiler pass to organise instructions into strands, identifying which values would be allocated to GPRs and which to accumulators.

The organisation of the ILDP architecture allows for future research into more flexible schemes of multiprocessing compared to the current approaches of symmetric multithreading and symmetric multiprocessing. Figure 6.1(a) shows a 4-way superscalar CPU, with the issue pipeline depicted as 'IP', and each functional unit represented as 'FU'. When a superscalar processor implements SMT, typically the issue logic will be replicated and then share the original execution resources (Figure 6.1(b)), while an SMP solution will replicate both issue and execution resources with no sharing (Figure 6.1(c)).

A multiprocessing ILDP system could take a hybrid approach, where two issue pipelines feed a set of e.g. twelve PEs – while the issue logic has been replicated twice, there is only 50% more execution logic. As some processing elements are dedicated to one issue pipeline and some are shared (rather than the fully shared approach of SMT or the fully dedicated approach of SMP), some of the performance issues present with SMT should be avoided, without the full cost of an SMP solution.

As the number of issue pipelines is increased, more interesting arrangements of execution resources could potentially be devised – e.g. the arrangement in Figure 6.2 with four issue units and 20 processing elements, where each issue unit has two dedicated PEs and two PEs shared with each other issue unit. This reduces the overall logic replication compared to a full 4-way SMP implementation, which would require 32 PEs, but limits the additional complexity of the

Figure 6.2 An example four-way multiprocessing ILDP CPU



PEs and the interconnection logic, as each PE need only communicate with a maximum of two of the issue pipelines and a limited subset of the other PEs. Performance should potentially be on par with a fully replicated SMP, as each issue pipeline has two dedicated PEs and need only contend with one other pipeline for each of the others – allocation strategies can be chosen to minimise conflicts as much as possible. This type of organisation would allow for very fine-grained power control, as individual processing elements and issue pipelines could be powered on or off, with the processor performance hopefully scaling well as the number of resources is increased or decreased. This is in marked contrast to current SMP solutions, which typically only allow power control at the level of an entire core.

Bibliography

[1] Vikas Agarwal, M S Hrishikesh, Stephen W Keckler, and Doug Burger.
Clock rate versus IPC: the end of the road for conventional microarchitectures.
In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259. IEEE Computer Society, 2000.
`http://citeseer.ist.psu.edu/agarwal00clock.html`
Referenced on page 36.

[2] Gerald Aigner, Amer Diwan, David L Heine, Monica S Lam, David L Moore, Brian R Murphy, and Constantine Sapuntzakis.
An overview of the SUIF2 compiler infrastructure.
`http://suif.stanford.edu/suif/suif2/doc-2.2.0-4/overview.ps`
Referenced on page 54.

[3] ARM Ltd.
An introduction to Thumb.
Technical Report ARM DVI-0001A, Advanced RISC Machines Ltd, March 1995
Referenced on page 33.

[4] Florin Baboescu and Dean M Tullsen.
Memory subsystem design for multithreaded processors.
Technical report, University of California, San Diego, 1997.
`http://citeseer.ist.psu.edu/577904.html`
Referenced on page 26.

[5] Doug Burger, Todd M Austin, and Steve Bennett.
Evaluating future microprocessors: The SimpleScalar tool set.
Technical Report CS-TR-1996-1308, University of Wisconsin-Madison, July 1996.
`http://citeseer.ist.psu.edu/burger96evaluating.html`
Referenced on pages 55 and 56.

[6] Doug Burger and James R Goodman.
Billion-transistor architectures.
IEEE Computer, volume 30 issue 9, pages 46–49, September 1997.
http://citeseer.ist.psu.edu/534110.html
Referenced on page 36.

[7] Stephen P Crago, Apoorv Srivastava, Kevin Obenland, and Alvin M Despain.
A high-performance, hierarchical decoupled architecture.
Technical Report ACAL-TR-96-07, Advanced Computer Architecture Laboratory, University of Southern California, November 1996.
http://citeseer.ist.psu.edu/crago96highperformance.html
Referenced on page 27.

[8] 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 *Proceedings of the International Symposium on Code Generation and Optimization*, pages 15–24. IEEE Computer Society, 2003.
citeseer.ist.psu.edu/dehnert03transmeta.html
Referenced on page 53.

[9] Richard Earnshaw.
Procedure call standard for the ARM architecture.
Technical Report GENC-003534, ARM, May 2006.
http://www.arm.com/miscPDFs/8031.pdf
Referenced on page 40.

[10] Susan J Eggers, Joel S Emer, Henry M Levy, Jack L Lo, Rebecca L Stamm, and Dean M Tullsen.
Simultaneous multithreading: A platform for next-generation processors.
IEEE Micro, volume 17 issue 5, pages 12–19, September/October 1997.
http://citeseer.ist.psu.edu/eggers97simultaneous.html
Referenced on page 25.

[11] Douglas J Evans.
The Cpu tools GENerator, CGEN, Mar 2001.
http://sources.redhat.com/cgen/docs-1.0/cgen.html
Referenced on page 54.

[12] Matthew K Farrens and Andrew R Pleszkun.
Overview of the PIPE processor implementation.
In *Proceedings of the 24th Annual Hawaii International Conference on System Sciences*, pages 433–443. IEEE Computer Society, January 1991.
http://citeseer.ist.psu.edu/farrens91overview.html
Referenced on page 27.

[13] Mark Game and Alan Booker.
Codepack: Code compression for PowerPC processors.
International Business Machines (IBM) Corporation, 1998
Referenced on page 33.

[14] A Gara, M A Blumrich, D Chen, G L-T Chiu, P Coteus, M E Giampapa, R A Haring, P Heidelberger, D Hoenicke, G V Kopcsay, T A Liebsch, M Ohmacht, B D Steinmacher-Burow, T Takken, and P Vranas.
Overview of the Blue Gene/L system architecture.
IBM Journal of Research and Development, volume 49 issue 2, pages 195–212, 2005.
`http://www.research.ibm.com/journal/rd/492/gara.pdf`
Referenced on page 29.

[15] Montse García, José González, and Antonio González.
Data caches for multithreaded processors.
In *Workshop on MultiThreaded Execution Architecture and Compilation*, January 2000.
`http://citeseer.ist.psu.edu/396929.html`
Referenced on page 26.

[16] Simcha Gochman, Avi Mendelson, Alon Naveh, and Efraim Rotem.
Introduction to Intel Core Duo processor architecture.
Intel Technology Journal, volume 10 issue 2, pages 89–98, May 2006.
`http://download.intel.com/technology/itj/2006/volume10issue02/vol10_art01.pdf`
Referenced on page 30.

[17] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown.
MiBench: A free, commercially representative embedded benchmark suite.
In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*. IEEE Computer Society, December 2001.
`http://www.eecs.umich.edu/mibench/Publications/MiBench.pdf`
Referenced on page 55.

[18] Linley Gwennap.
Digital 21264 sets new standard.
Microprocessor Report, volume 10 issue 14, pages 11–16, October 1996.
`http://www.cs.virginia.edu/~dp8x/alpha21264/paper/new%20standard.pdf`
Referenced on page 28.

[19] Allan M Hartstein and Thomas R Puzak.
The optimum pipeline depth for a microprocessor.
In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 7–13. IEEE Computer Society, May 2002.
`http://systems.cs.colorado.edu/ISCA2002/FinalPapers/hartsteina_optimum_pipeline_color.pdf`
Referenced on page 35.

[20] John L Henning.
SPEC CPU2000: Measuring CPU performance in the new millennium.
IEEE Computer, volume 33 issue 7, pages 28–35, July 2000.
`http://www.spec.org/cpu2000/papers/COMPUTER_200007.JLH.pdf`
Referenced on page 55.

[21] S Hily and A Seznec.
Standard memory hierarchy does not fit simultaneous multithreading.
In *Workshop on MultiThreaded Execution Architecture and Compilation*, January 1998.
`http://citeseer.ist.psu.edu/hily98standard.html`
Referenced on page 26.

[22] Sébastien Hily and André Seznec.
Contention on 2nd level cache may limit the effectiveness of simultaneous multithreading.
Technical Report PI-1086, Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA), 1997.
`http://citeseer.ist.psu.edu/hily97contention.html`
Referenced on page 26.

[23] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel.
The microarchitecture of the Pentium 4 processor.
Intel Technology Journal, volume 5 issue 1, February 2001.
`http://www.intel.com/technology/itj/q12001/pdf/art_2.pdf`
Referenced on pages 31 and 35.

[24] M S Hrishikesh, Norman P Jouppi, Keith I Farkas, Doug Burger, Stephen W Keckler, and Premkishore Shivakumar.
The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays.
In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 14–24. IEEE Computer Society, May 2002.
`http://systems.cs.colorado.edu/ISCA2002/FinalPapers/hrishikeshm_optimal_revised.ps`
Referenced on page 35.

[25] Xianglong Huang, J Eliot B Moss, Kathryn S McKinley, Steve Blackburn, and Doug Burger.
Dynamic Simplescalar: Simulating Java virtual machines.
Technical Report TR-03-03, University of Texas, 2003.
`http://www.cs.utexas.edu/ftp/pub/techreports/tr03-03.ps.gz`
Referenced on page 56.

[26] IBM Microelectronics Division.
The PowerPC 405 Core, February 1998.
`http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF77852569970063B427/$file/405cr_wp.pdf`
Referenced on page 56.

[27] IBM Microelectronics Division.
The PowerPC 440 Core, September 1999.
`http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF77852569970063431C/$file/440_wp.pdf`
Referenced on page 56.

[28] ITRS.
International Technology Roadmap for Semiconductors: Interconnect, 2006.
`http://www.itrs.net/Links/2006Update/FinalToPost/09_Interconnect2006Update.pdf`
Referenced on page 17.

[29] Robert M Keller.
Look-ahead processors.
ACM Computing Surveys, volume 7 issue 4, pages 177–195, 1975.
`http://portal.acm.org/citation.cfm?id=356657`
Referenced on page 23.

[30] Ho-Seop Kim and James E Smith.
An instruction set and microarchitecture for instruction level distributed processing.
In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 71–81. IEEE Computer Society, May 2002.
`http://citeseer.ist.psu.edu/kim02instruction.html`
Referenced on pages 15, 28, 37, 39, 45, 46, 49, 50, 53, 55, 56, 64, 78, 80, and 116.

[31] Nam Sung Kim, Todd Austin, David Blaauw, Trevor Mudge, Kristián Flautner, Jie S Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan.
Leakage current: Moore's law meets static power.
IEEE Computer, volume 36 issue 12, pages 68–75, 2003
Referenced on page 34.

[32] K Kissell.
MIPS16: High-density MIPS for the embedded market.
Silicon Graphics MIPS Group, 1997
Referenced on page 33.

[33] Alexander Klaiber.
The technology behind Crusoe processors: Low-power x86-compatible processors implemented with Code Morphing software.
Technical report, Transmeta Corporation, 2000.
http://www.transmeta.com/pdfs/paper_aklaiber_19jan00.pdf
Referenced on page 53.

[34] Ronny Krashinsky and Michael Sung.
Decoupled architectures for complexity-effective general purpose processors.
http://citeseer.ist.psu.edu/krashinsky00decoupled.html.
Advanced VLSI Computer Architecture term project, Massachusetts Institute of Technology, December 2000
Referenced on page 27.

[35] Alexei Kudriavtsev and Peter Kogge.
SMT possibilities for decoupled architecture.
In *Technical Committee on Computer Architecture (TCCA) Newsletter: Papers from MEmory access DEcoupling for superscalar and multiple issue Architectures (MEDEA-2000) Workshop*. IEEE Computer Society, January 2001.
http://tab.computer.org/tcca/NEWS/jan2001/kudriavtsev.pdf
Referenced on page 27.

[36] Mikko H Lipasti, Brian R Mestan, and Erika Gunadi.
Physical register inlining.
In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 325–335. IEEE Computer Society, June 2004.
http://www.ece.wisc.edu/~pharm/papers/isca2004_egunadi.pdf
Referenced on page 109.

[37] Jack L Lo, Susan J Eggers, Joel S Emer, Henry M Levy, Rebecca L Stamm, and Dean M Tullsen.
Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading.
ACM Transactions on Computer Systems, volume 15 issue 3, pages 322–354, August 1997.
http://www.cs.washington.edu/research/smt/papers/tlp2ilp.final.pdf
Referenced on page 25.

[38] Deborah T Marr, Frank Binns, David L Hill, Glenn Hinton, David A Koufaty, J Alan Miller, and Michael Upton.
Hyper-threading technology architecture and microarchitecture.
Intel Technology Journal, volume 6 issue 1, pages 4–15, February 2002.
`http://download.intel.com/technology/itj/2002/volume06issue01/art01_hyper/vol6iss1_art01.pdf`
Referenced on page 26.

[39] MIPS Technologies, Inc.
The MIPS MT application-specific extension to the MIPS32 architecture.
Technical Report MD00378, 2005
Referenced on page 26.

[40] Hans-Peter Nilsson.
Porting GCC for dunces, May 2000.
`http://ftp.axis.se/pub/users/hp/pgccfd/pgccfd.pdf`
Referenced on page 54.

[41] Thomas Pabst and Frank Völkel.
Hot Spot: How modern processors cope with heat emergencies.
Tom's Hardware Guide, September 2001.
`http://tomshardware.co.uk/2001/09/17/hot_spot/`
Referenced on page 34.

[42] Subbarao Palacharla.
Complexity-effective superscalar processors.
PhD thesis, University of Wisconsin-Madison, 1998.
`http://www.ece.wisc.edu/~jes/papers/subba.thesis.pdf`
Referenced on pages 27, 28, 46, 76, 105, and 112.

[43] David A Patterson and David R Ditzel.
The case for the reduced instruction set computer.
SIGARCH Computer Architecture News, volume 8 issue 6, pages 25–33, 1980.
`http://portal.acm.org/citation.cfm?id=641917`
Referenced on pages 30 and 31.

[44] Richard Phelan.
Improving ARM code density and performance.
Technical report, ARM, June 2003.
`http://www.arm.com/pdfs/Thumb-2CoreTechnologyWhitepaper-Final4.pdf`
Referenced on page 33.

[45] Norman Ramsey and Mary F Fernández.
The New Jersey machine-code toolkit.
In *Proceedings of the 1995 USENIX Technical Conference*, pages 289–302. USENIX Association, January 1995.
`http://www.eecs.harvard.edu/~nr/pubs/tk-usenix.ps`
Referenced on page 54.

[46] Norman Ramsey and Mary F Fernández.
Specifying representations of machine instructions.
ACM Transactions on Programming Languages and Systems, volume 19 issue 3, pages 492–524, May 1997.
`http://www.eecs.harvard.edu/~nr/pubs/specifying.ps`
Referenced on page 54.

[47] James E Smith.
Decoupled access/execute computer architectures.
In *Proceedings of the 9th Annual International Symposium on Computer Architecture*, pages 231–238. IEEE Computer Society, 1982.
`http://www.cs.berkeley.edu/~yatish/prelim/daeca.pdf`
Referenced on pages 26 and 101.

[48] James E Smith.
Characterizing computer performance with a single number.
Communications of the ACM, volume 31 issue 10, pages 1202–1206, 1988
Referenced on page 17.

[49] James E Smith.
Instruction level distributed processing: Adapting to shifting technology.
In *Proceedings of the 7th International Conference on High Performance Computing*, pages 245–258. Springer, December 2000.
`http://www.ece.wisc.edu/~jes/papers/hipc.00.pdf`
Referenced on page 28.

[50] James E Smith.
Instruction-level distributed processing.
IEEE Computer, volume 34 issue 4, pages 59–65, April 2001.
`http://citeseer.ist.psu.edu/534110.html`
Referenced on page 28.

[51] James E Smith and Andrew R Pleszkun.
Implementation of precise interrupts in pipelined processors.
In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 36–44, Los Alamitos, CA, USA, 1985. IEEE Computer Society.
`http://portal.acm.org/citation.cfm?id=327125`
Referenced on page 23.

[52] James E Smith and Gurindar S Sohi.
The microarchitecture of superscalar processors.
Proceedings of the IEEE, volume 83, pages 1609–1624, 1995.
`http://citeseer.ist.psu.edu/35243.html`
Referenced on page 23.

[53] Michael D Smith and Glenn Holloway.
An introduction to Machine SUIF.
`http://www.eecs.harvard.edu/hube/software/nci/overview.html`
Referenced on page 54.

[54] Michael D Smith, Mike Johnson, and Mark A Horowitz.
Limits on multiple instruction issue.
In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pages 290–302, New York, NY, 1989. ACM Press.
`http://citeseer.ist.psu.edu/smith89limits.html`
Referenced on page 25.

[55] Gurindar S Sohi and Sriram Vajapeyam.
Instruction issue logic for high-performance, interruptable pipelined processors.
In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 27–34. IEEE Computer Society, 1987.
`http://www.eecs.harvard.edu/~dbrooks/cs146/sohi-ruu.pdf`
Referenced on page 24.

[56] Michael Sung, Ronny Krashinsky, and Krste Asanović.
Multithreading decoupled architectures for complexity-effective general purpose computing.
SIGARCH Computer Architecture News, volume 29 issue 5, pages 56–61, 2001.
`http://citeseer.ist.psu.edu/sung01multithreading.html`
Referenced on page 27.

[57] Deependra Talla and Lizy K John.
Mediabreeze: A decoupled architecture for accelerating multimedia applications.
SIGARCH Computer Architecture News, volume 29 issue 5, pages 62–67, 2001.
`http://citeseer.ist.psu.edu/526028.html`
Referenced on page 27.

[58] R M Tomasulo.
An efficient algorithm for exploiting multiple arithmetic units.
IBM Journal of Research and Development, volume 11 issue 1, pages 25–33, 1967.
`http://www.research.ibm.com/journal/rd/111/tomasulo.pdf`
Referenced on page 116.

[59] Dean M Tullsen, Susan Eggers, and Henry M Levy.
Simultaneous multithreading: Maximizing on-chip parallelism.
In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. IEEE Computer Society, 1995.
http://citeseer.ist.psu.edu/tullsen95simultaneous.html
Referenced on page 25.

[60] Gary Tyson, Matthew Farrens, and Andrew R Pleszkun.
MISC: a multiple instruction stream computer.
In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO 25)*, pages 193–196. IEEE Computer Society, 1992.
http://citeseer.ist.psu.edu/tyson92misc.html
Referenced on page 27.

[61] David W Wall.
Limits of instruction-level parallelism.
In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pages 176–189, New York, NY, 1991. ACM Press.
http://citeseer.ist.psu.edu/wall90limits.html
Referenced on page 25.

[62] Panit Watcharawitch and Simon Moore.
JMA: The Java-Multithreading Architecture for embedded processors.
In *International Conference on Computer Design (ICCD)*. IEEE Computer Society, September 2002.
http://www.srcf.ucam.org/~pw240/Project/JMA_long.pdf
Referenced on page 25.

[63] Panit Watcharawitch and Simon Moore.
MulTEP: MultiThreaded Embedded Processors.
In *An International Symposium on Low-Power and High-Speed Chips (Cool Chips) IV*, volume I. IEEE/IEICE/IPSJ/ACM SIGARCH Computer Society, April 2003.
http://www.cl.cam.ac.uk/~pw240/Coolchip_MulTEP.pdf
Referenced on page 25.

[64] Neil H E Weste and David Harris.
CMOS VLSI design: A Circuits and Systems Perspective.
Pearson/Addison-Wesley, 3rd edition, 2005
Referenced on pages 32 and 34.