Compiler-Directed Energy Savings in Superscalar Processors

Timothy M. Jones

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2006

Abstract

Superscalar processors contain large, complex structures to hold data and instructions as they wait to be executed. However, many of these structures consume large amounts of energy, making them hotspots requiring sophisticated cooling systems. With the trend towards larger, more complex processors, this will become more of a problem, having important implications for future technology.

This thesis uses compiler-based optimisation schemes to target the issue queue and register file. These are two of the most energy consuming structures in the processor. The algorithms and hardware techniques developed in this work dynamically adapt the processor's resources to the changing program phases, turning off parts of each structure when they are unused to save dynamic and static energy.

To optimise the issue queue, the compiler analysis tracks data dependences through each program procedure. It identifies the critical path through each program region and informs the hardware of the minimum number of queue entries required to prevent it slowing down. This reduces the occupancy of the queue and increases the opportunities to save energy. With just a 1.3% performance loss, 26% dynamic and 32% static energy savings are achieved.

Registers can be idle for many cycles after they are last read, before they are released and put back on the free-list to be reused by another instruction. Alternatively, they can be turned off for energy savings. Early register releasing can be used to perform this operation sooner than usual, but hardware schemes must wait for the instruction redefining the relevant logical register to enter the pipeline. This thesis presents an exploration of compiler-directed early register releasing. The compiler can exactly identify the last use of each register and pass the information to the hardware, based on simple data-flow and liveness analysis. The best scheme achieves 15% dynamic and 19% static energy savings.

Finally, the issue queue limiting and early register releasing schemes are combined for energy savings in both processor structures. Four different configurations are evaluated bringing 25% to 31% dynamic and 19% to 34% static issue queue energy savings and reductions of 18% to 25% dynamic and 20% to 21% static energy in the register file.

Acknowledgements

I would like to thank first and foremost my supervisor, Dr. Michael O'Boyle, for all the support and advice he has given over the past three years. I would also like to thank everyone in Barcelona with whom I have worked over this time, especially Professor Antonio González for his invaluable insights into processor architectures, Jaume Abella for his patience in answering all my mundane questions, Oğuz Ergin for microarchitecture design discussions, and Enric Gibert for friendship, conversations and lunches.

In Edinburgh, I'd like to thank everyone in the compiler and architecture design group for interest, advice and support. In no particular order they are Marcelo, Björn, Tom, Shun, Grigori, John, Aris, Constantino, Jialin, Chris, John and all other members. I'd especially like to thank my office mates, Allan and Andy, for all discussions, lunches, tea and company, and Rick for coffee-room chats. I'd also like to thank everyone who took the time to read over chapters of this thesis, providing useful feedback, even with limited knowledge of computer science.

I'd like to thank everyone outside Informatics who has made my last three years so enjoyable. Especially my flatmates past and present (Kat, Dustin, Maarika, Sonya, Carla) and friends from orchestras, bands and chamber groups, too numerous to mention here.

Last, and by no means least, thanks to my parents and brothers for the constant support and interest in my work whilst I've been studying.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification. Some of the material used in this thesis has been published in the following papers:

- Timothy M. Jones, Michael F.P. O'Boyle, Jaume Abella, and Antonio González. Software directed issue queue power reduction. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*, 2005.
- Timothy M. Jones, Michael F.P. O'Boyle, Jaume Abella, Antonio González, and Oğuz Ergin. Compiler directed early register release. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2005.

To my family.

Table of Contents

1 Introduction **1**

1.1 The Problem 1

1.2 Contributions 2

1.3 Structure 3

2 Superscalar Microarchitecture **5**

2.1 Superscalar Processors 5

2.2 Instruction Dispatch and Issue 7

2.2.1 Type of Issue Queue 7

2.2.2 Operand Fetching 8

2.2.3 Instruction Wakeup 9

2.2.4 Banked Issue Queue 10

2.2.5 Sources of Energy Consumption 11

2.3 Register Renaming 12

2.3.1 Separate Rename Buffers 14

2.3.2 Combined Reorder Buffer 15

2.3.3 Centralised Register File 16

2.3.4 Map Tables 17

2.3.5 Banked Register File 18

2.3.6 Sources of Energy Consumption 19

2.4 Summary 19

3 Related Work **21**

3.1 Issue Queue 21

3.1.1 Reducing Wakeups 22

3.1.2 Gating Pipeline Stages . . . 23
3.1.3 Latency Calculation . . . 26
3.1.4 Dependence Tracking . . . 28
3.1.5 Compiler Schemes . . . 30
3.2 Register File . . . 32
3.2.1 Value Optimisation . . . 33
3.2.2 Late Allocation and Early Release . . . 35
3.2.3 Ports Reduction . . . 38
3.2.4 Register Caching . . . 42
3.2.5 Compiler Schemes . . . 44
3.3 Summary . . . 46

4 Infrastructure **47**
4.1 Benchmarks . . . 47
4.2 Compiler . . . 48
4.3 Simulation Environment . . . 49
4.3.1 Simulator . . . 49
4.3.2 Energy Measurements . . . 50
4.4 Summary . . . 53

5 Issue Queue Energy Savings **55**
5.1 Motivation . . . 56
5.2 Microarchitecture . . . 58
5.2.1 Issue queue . . . 58
5.2.2 Fetch queue . . . 59
5.2.3 Register Files . . . 59
5.3 Compiler Analysis . . . 60
5.3.1 Program Representation . . . 60
5.3.2 Critical Path Model . . . 61
5.3.3 Specialised DAG Analysis . . . 64
5.3.4 Specialised Loop Analysis . . . 68
5.4 Coarse-grained Throttling . . . 71
5.4.1 Issue Queue . . . 72
5.4.2 Procedure NOOPs . . . 72

5.4.3 Tags . . . 73
5.5 Fine-grained Throttling . . . 74
5.5.1 Issue Queue . . . 75
5.5.2 Block NOOPs . . . 76
5.5.3 Tags . . . 77
5.6 Summary . . . 79

6 Register File Energy Savings **81**
6.1 Motivation . . . 82
6.1.1 Early Register Releasing . . . 82
6.1.2 Hardware Oracle versus Compiler Analysis . . . 85
6.2 Microarchitecture . . . 88
6.2.1 Checkpointed Register File . . . 88
6.2.2 Early Releasing . . . 89
6.2.3 Interrupts and Exceptions . . . 90
6.3 Commit Releasing . . . 90
6.3.1 Commit NOOPs . . . 91
6.3.2 Branches . . . 93
6.3.3 Procedure Boundaries . . . 94
6.3.4 Instruction Tagging . . . 95
6.3.5 Combined . . . 96
6.3.6 Results . . . 96
6.4 Issue Releasing . . . 98
6.4.1 One-use Registers . . . 99
6.4.2 Two-use Registers . . . 103
6.4.3 Inter-procedural Analysis . . . 105
6.4.4 Results . . . 105
6.5 Combined Approaches . . . 109
6.5.1 Results . . . 110
6.6 Register File Size Sensitivity . . . 113
6.7 Summary . . . 115

7 Combined Optimisations **117**
7.1 Motivation . . . 117

7.2 Differences in Architectures . . . 118
7.3 Results . . . 118
7.3.1 Minimal Hardware . . . 119
7.3.2 No ISA . . . 121
7.3.3 Lowest Occupancy . . . 123
7.3.4 No NOOPs . . . 126
7.4 Summary . . . 129

8 Conclusions **131**
8.1 Contributions . . . 131
8.1.1 Issue Queue . . . 131
8.1.2 Register File . . . 132
8.2 Critical Analysis . . . 133
8.3 Future Work . . . 136

A Energy Breakdown **137**

Bibliography **139**

Chapter 1

Introduction

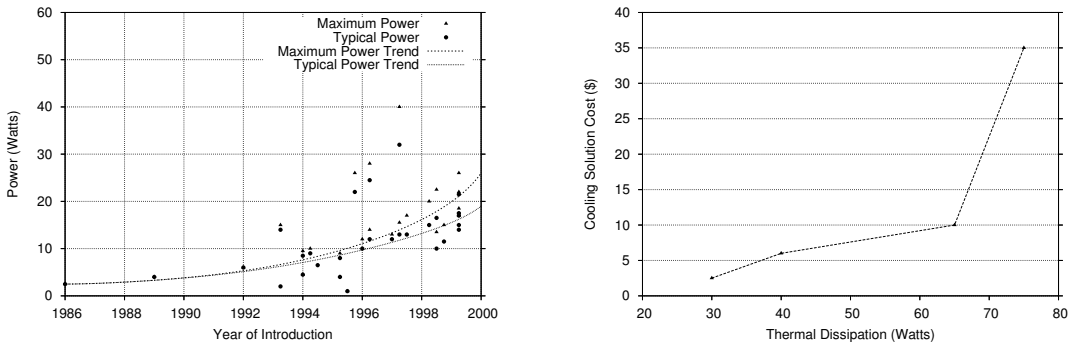
This thesis investigates compiler-directed schemes to reduce the energy consumed in a superscalar processor. This chapter briefly describes the problem and contributions made, motivating the work.

1.1 The Problem

Superscalar processors contain complex logic and structures to hold instructions and information as they pass through the pipeline. In recent years, an increase in the number of in-flight instructions present within the processor has contributed to greater performance. This has been achieved by increasing the size of the internal structures and the number of instructions fetched into the processor on each clock cycle. Increases in clock frequency have also contributed to an increase in instruction throughput.

However, greater processor performance achieved through these means comes at the price of greater energy requirements and power dissipation as the energy is converted into heat. This causes problems for circuit designers who must incorporate large and expensive cooling systems into their architectures to avoid failure. This problem will only exacerbate if current trends continue and processors approach the 1 billion transistor mark.

Figure 1.1 shows the trends in power dissipation of Intel processors introduced over a fifteen year period and how the cost of cooling has increased as the amount of heat produced has risen, both taken from Gunther *et al*. [36]. As figure 1.1(a) shows, more recent processors have a much higher maximum power dissipation, increasing by a factor of just over 2 every four years [36]. Figure 1.1(b) shows the costs involved in removing this power (converted to heat)



(a) Power trends **(b) Cooling cost**

Figure 1.1. Trends in power dissipation of Intel microprocessors and the cost of cooling them [36].

from the system. It can be seen that the cooling cost rises non-linearly with the power of the processor [36]. These graphs show that reducing the amount of power a processor dissipates, or the amount of energy it consumes, directly benefits its cost of production.

Within the processor, different structures consume varying amounts of energy. Folegnani and González quantified the energy contributions for each structure in their processor and the six structures consuming the most are shown in table 1.1 [32]. In their processor, the reorder buffer was used to hold results from executed instructions instead of using a separate register file (see chapter 2). The reading and writing activity to access this data was responsible for the high energy consumption of the reorder buffer [32]. When a register file is used, as assumed in this thesis, it is the second most energy consuming structure, making the reorder buffer's contribution smaller.

Table 1.1 shows clearly that the two most energy consuming structures in this processor are the issue queue and reorder buffer (register file when separated). Therefore, reducing their energy consumption is more beneficial to the processor than targeting anything else. Hence, the main focus of this thesis is energy reduction targeted at the issue queue and register file.

1.2 Contributions

This thesis presents schemes that reduce the energy requirements of the issue queue and register file with little impact on performance.

Firstly, a compiler-directed scheme is presented that dynamically limits the number of en-

Reorder Buffer	Issue Queue	Rename Table	L2 Cache	I-Cache	D-Cache
27.1%	26.5%	13.8%	11.1%	5.4%	5.0%

Table 1.1. Structures consuming the greatest amounts of processor energy for integer benchmarks [32].

tries in the issue queue allowed to contain instructions, reducing its energy consumption whilst having little impact on performance. Unlike other approaches, this scheme uses knowledge of the future issue queue requirements of each program to guide the throttling, rather than basing it on past performance. Thus, the work presented in this thesis can adapt more rapidly to the changing program requirements and the decisions made are more reliable, resulting in smaller performance losses.

Secondly, several techniques are presented that release registers much earlier than usual, allowing better utilisation of the register file, reducing register pressure and saving energy. The schemes presented in this thesis can release registers far earlier than hardware-based early releasing approaches through the use of compiler-inferred knowledge of future register requirements. An oracle is developed to determine the maximum savings possible and used as a comparison for the proposed techniques, further verifying the quality of these approaches.

Finally, due to the orthogonal nature of the issue queue limiting and early register releasing schemes, several of the proposed compiler-directed approaches are combined for increased overall processor energy savings.

1.3 Structure

This thesis is structured as follows. Chapter 2 describes the pipeline of a superscalar processor. In particular, it focuses on the dispatch, issue and renaming of instructions as they directly affect the issue queue and register file, the two structures that are targeted in this thesis. Sources of energy consumption are considered and the method of banking each structure is described.

Chapter 3 describes work related to this thesis that optimises the issue logic and register file. There are schemes that attempt to limit the number of wakeups in the issue queue whereas other approaches remove the wakeup logic altogether, replacing it with schemes taking advantage of the dependences between instructions. In the register file, various banking techniques are proposed along with schemes to make more efficient use of the registers. Other compiler-directed approaches that target the issue queue or register file are also presented.

The benchmark programs used to evaluate the proposed schemes in this thesis are presented

in chapter 4 along with the compiler and simulation environment. Chapter 5 then describes a scheme to dynamically limit the number of instructions entering the issue queue, increasing the number of free entries which can be turned off for energy savings. The microarchitecture changes are described along with the compiler analysis and implementation. These are based on critical path analysis specialised for directed acyclic graphs and loops. Two variations of dynamic throttling are evaluated using both tags and special NOOPs to pass information to the processor from the compiler. Results show high energy savings with little performance degradation, out-performing previous hardware-based schemes.

Optimising the register file is the subject of chapter 6. Here, a hardware oracle is implemented to determine the extent of register pressure savings achievable by releasing registers earlier than usual. The limits of compiler-directed approaches are also considered and it is found that significant savings can be made beyond those already proposed in the literature. Several compiler-directed schemes are presented that release registers at the issue and commit pipeline stages using special NOOPs, instruction tagging and register renaming. These are then combined to produce schemes that reduce the register file's energy requirements when using a large number of registers, and increase performance with a small number, out-performing previous hardware and software-based approaches.

Chapter 7 presents results from combining schemes presented in chapters 5 and 6. These show greater processor energy savings than when they are used in isolation.

Finally, chapter 8 concludes this thesis, summarising the main results and contributions, providing a critical analysis of the schemes presented and describing directions future work could pursue.

Chapter 2

Superscalar Microarchitecture

This chapter describes a baseline superscalar microarchitecture, as used in this thesis. Please see Hennessy and Patterson [37] for a more in-depth account. This chapter describes the pipeline of a superscalar processor with emphasis on the issue queue and register file. As described in chapter 1, these are the structures with the highest energy consumption within the processor and are thus the target of this thesis.

This chapter is structured as follows. Section 2.1 gives an overview of a superscalar processor pipeline, describing the flow of instructions and data. Section 2.2 then focuses on instruction dispatch and issue, explaining the issue queue structure and the process of instruction wakeup. Section 2.3 describes the register file and implementations of register renaming and finally section 2.4 summarises this chapter.

2.1 Superscalar Processors

Modern superscalar processors achieve high performance by executing multiple instructions out-of-order every cycle. Hardware inside the processor detects independent instructions and allows them to execute in parallel. This is often referred to as the *instruction-level parallelism* (ILP) that the processor exploits within the program.

An example of a superscalar pipeline is shown in figure 2.1. In this diagram, the flow of instructions is represented by arrows with solid lines, whereas the flow of data is represented by arrows with dashed lines. Every cycle a number of instructions enter the processor and make their way through various stages of the pipeline. Instructions are said to be *fetched* when they are brought into the pipeline from the instruction cache. Similarly they are *dispatched* when

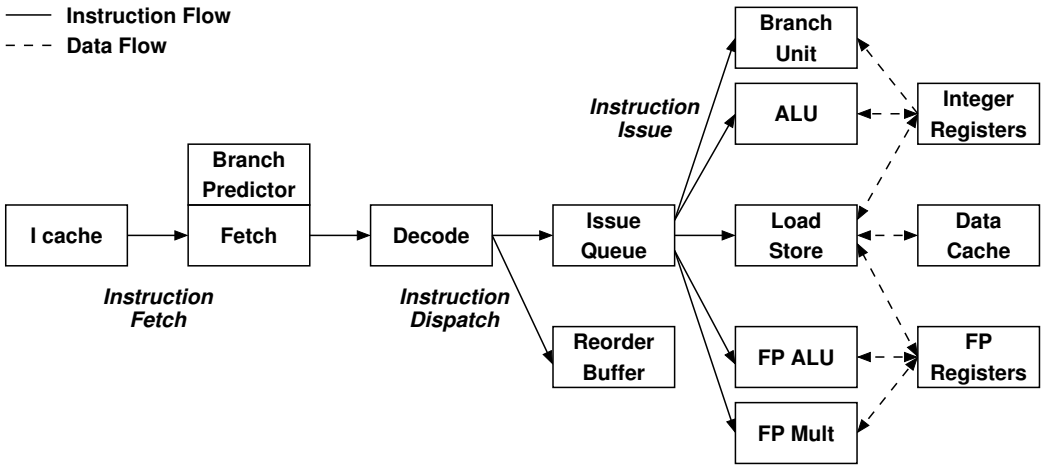


Figure 2.1. An example superscalar pipeline showing flow of instructions from the instruction cache to functional units (arrows with solid lines) and flow of data between functional units, registers and the data cache (arrows with dashed lines). Instruction fetch is the process of bring instructions from the instruction cache into the pipeline and dispatch is when they are moved into the issue queue and reorder buffer. Instructions are issued when they are sent to a functional unit to be executed.

they move to the issue queue and reorder buffer, and *issued* when they are sent to a functional unit for execution. The number of instructions dispatched each cycle is known as the *dispatch width* and the number issued the *issue width*.

Referring to figure 2.1, an instruction's journey begins at the first level instruction cache from where it is brought into the processor and placed in the fetch queue. On a control transfer instruction (branch, procedure call, etc.), the next instruction is speculated by some branch prediction logic to prevent the pipeline from stalling until the correct next instruction is known, which could be many cycles later. If the prediction is correct, the processor gains performance from having already fetched later instructions into the pipeline. On a mis-prediction, later instructions must be squashed and their resources freed. Fetch can then restart with the correct next instruction. This means work performed in executing the squashed instructions is wasted, but the infrequent nature of branch mis-predictions combined with the performance gains made through its usage make this worthwhile.

After decoding, instructions are dispatched to the issue queue and reorder buffer. To help facilitate out-of-order execution and overcome the limitations on register numbers imposed by

the instruction set architecture (ISA), each instruction is allocated a new, unique location to store its result (called a rename buffer) in a process known as register renaming. To preserve the dependences between instructions, each source operand is also renamed to the location containing its producer's result. For more information on register renaming see section 2.3.

Instructions can enter the issue queue before the values of their source operands have been calculated. Therefore, they must wait here until their input data is available, at which time they can be issued to a functional unit for execution. During issue, each instruction reads its source operands from the register file and after execution writes its result back, or in the case of a store, sends it to the data cache.

Instructions are executed out-of-order and any updates made to the processor's state (e.g. writes to registers or memory) are made in temporary locations (e.g. the rename buffers) until the processor can guarantee that the instructions are non-speculative and there are no interrupts or exceptions to be serviced. Once these conditions are satisfied, instructions can leave the processor in the order they entered, which is called *committing*. The reorder buffer facilitates this process by holding instructions in the order they were dispatched to the issue queue. Instructions cannot then commit until they are the oldest instruction contained in the reorder buffer and they have finished execution. At the same time as they commit, the changes they make to the processor's state become permanent.

2.2 Instruction Dispatch and Issue

The issue queue holds instructions whilst they wait for their source operands to be calculated. It also allows out-of-order execution of instructions. An early version was proposed by Tomasulo to allow dynamic scheduling in the presence of false data dependences in the IBM 390/91 processor [82]. This section discusses various implementations of the issue queue with reasons for its high energy consumption. The structure of the queue is described in section 2.2.1 and operand fetching schemes discussed in section 2.2.2. Section 2.2.3 describes the process of instruction wakeup, section 2.2.4 describes the issue queue banking scheme used in this thesis, and finally section 2.2.5 discusses sources of energy consumption within the issue queue.

2.2.1 Type of Issue Queue

The issue queue can be a single structure holding all instructions (sometimes split into integer and floating point queues), such as that employed in the PentiumPro (1995), or there can be a

number of smaller issue queues. In the latter case, used in the PowerPC and AMD K5, each queue is usually called a reservation station and is connected to either a single functional unit or a group of them.

When using a reservation station for each functional unit, instruction issue consists of choosing one ready instruction from each station and forwarding it to the functional unit for execution. Where a group of functional units share a reservation station, the issue logic must be able to select several ready instructions each cycle. With a single issue queue, the scheduler is more complex, needing to select and issue a whole issue width's worth of instructions each cycle.

There are several ways to implement an issue queue. The first is as a circular buffer where instructions are added in program order at the tail, which moves round until it reaches the head pointer. Then dispatch stops until the instruction at the head has issued, when it can continue again. As instructions issue out-of-order from the middle of the queue, holes appear. The designer can choose to implement a compaction scheme where instructions are shifted towards the head to fill these empty entries. However, this increases the complexity of the queue and consumes a large amount of energy moving instructions each cycle. The benefit of compaction is that the queue resources are used more efficiently.

In the second scheme, instructions can be placed anywhere in the queue that there is a free entry. This also allows for better utilisation of the queue but makes the selection of ready instructions harder, due to the fact that it is preferable to schedule older instructions first. These are chosen over younger ones to prevent them being the bottleneck to commit and to allow their resources to be recycled quickly.

In summary, a single issue queue is more complex to implement than single reservation stations, but provides greater flexibility to the processor in extracting ILP. Hence, in this thesis a single issue queue is used for both integer and floating point instructions. More information on this queue can be found in chapter 4.

2.2.2 Operand Fetching

In this thesis, a RISC (reduced instruction set computing) architecture is used, where each instruction has a maximum of two source operands. These can be obtained in two ways: either at dispatch or at issue.

When dispatch-bound operand fetching is employed, the issue queue stores the data that each instruction will operate on and must be large enough to accommodate this. When issue-

bound operand fetching is used, space is not needed to store operand values, hence the issue queue can be smaller.

One downside to using issue-bound operand fetching is that extra register file ports may be needed to read data. This is because, in some processors, such as the PentiumPro, AMD K5 and Sparc 64, the issue width is greater than the dispatch width. Complex instructions (e.g. divide and square root) cannot always be perfectly pipelined, so in order to maintain a balance between instruction dispatch and issue, the issue width should be slightly higher [74]. By reading registers at issue when this is the case, more read ports are needed into the register file than when reading registers at dispatch, increasing its size and access time complexity.

In summary, neither method gives clear benefits and there are examples of processors using either scheme [74]. In this thesis, optimisations presented in chapter 5 that target the issue queue use dispatch-bound operand fetching because the scheme used for comparison employs this method. In chapter 6, where early register releasing is considered, issue-bound fetching is used for the same reasons. Finally, chapter 7 optimises both the issue queue and the register file and results presented in that chapter are evaluated with both schemes.

2.2.3 Instruction Wakeup

When an instruction dispatches, the register file is checked to determine the availability of each source operand. A tag is saved in the issue queue for each unavailable operand, which is simply the physical register number. When an instruction finishes execution it broadcasts the tag (physical register number) of its destination register to the issue queue. Instruction wakeup consists of checking these returning tags with the tags of those instructions waiting in the queue. Upon a match, the relevant operand can be marked as *Ready* and when both source operands are in this state the instruction can be issued to a functional unit. Figure 2.2 shows a block diagram of the wakeup logic for a single instruction.

Each cycle the returning tags are broadcast to the queue along the taglines, running vertically in the diagram. The comparators check for a match with the stored tag and an *OR* is performed on the result. Only one returning tag will ever match an operand's stored tag because they represent physical registers and no two in-flight instructions can have the same physical register as their result's destination. If there is a match, the *Ready* bit for the relevant operand is set. In addition to this, when dispatch-bound operand fetching is used, the actual result value is also written into the issue queue. Details of the energy consumption during wakeup can be found in section 2.2.5.

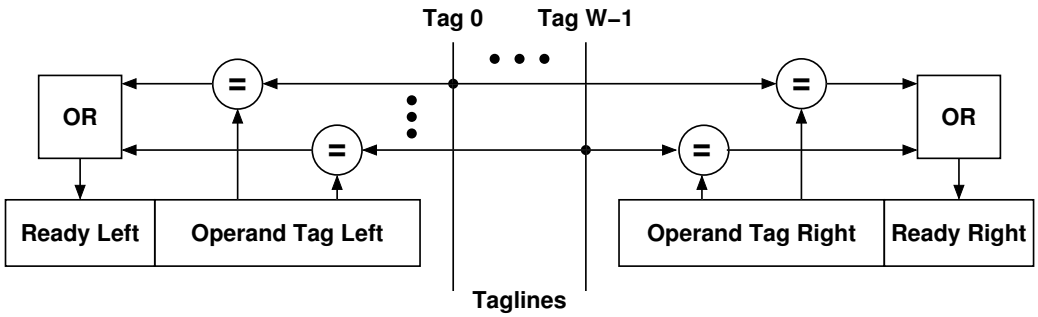


Figure 2.2. Issue logic for matching tags with issue width *W* [32].

2.2.4 Banked Issue Queue

By organising the issue queue entries into independent banks, greater flexibility can be given to energy saving schemes. Each bank can be turned off when it contains no valid data, to save dynamic energy during reads, writes and wakeups, and static energy for the duration it is off.

The banked issue queue used in this thesis is based on a design by Buyuktosunoglu *et al.* [21]. Figure 2.3 shows a block diagram of the queue logic for an issue queue of four banks. There are three main parts: the CAM (content addressable memory) cells, which store the source operand tags; the RAM (random access memory) cells, where all other instruction data (opcode, source operand data and destination tag) is held; and the selection logic.

CAM cells are used for the source operand tags because they provide the ability to match the stored data with the returning tags during instruction wakeup (see section 2.2.3). However, RAM cells are smaller and less power-hungry, so are used for the rest of the issue queue data because it does not need the matching capabilities provided by the CAMs. The selection logic sits between the two banks to be close to the *Ready* bits for each operand (stored in the CAM cells) and the instruction itself (stored in the RAMs). Each bank of CAM cells can be turned off independently of all others, with its corresponding RAM bank turned off at the same time. The selection logic is always on but this consumes less energy than the wakeup logic [67].

To avoid a delay in the wakeup logic, each tag has a global tagline [21]. An *AND* is made with the *Enable* signal and the output drives local taglines in each CAM bank. The *Enable* signal for each bank is a simple *OR* of the *Valid* bits for each entry in the bank. Empty entries are invalid so entries within an empty bank do not have their *Valid* bits on and thus the bank's *Enable* bit is also off, guaranteeing that only completely empty banks are turned off.

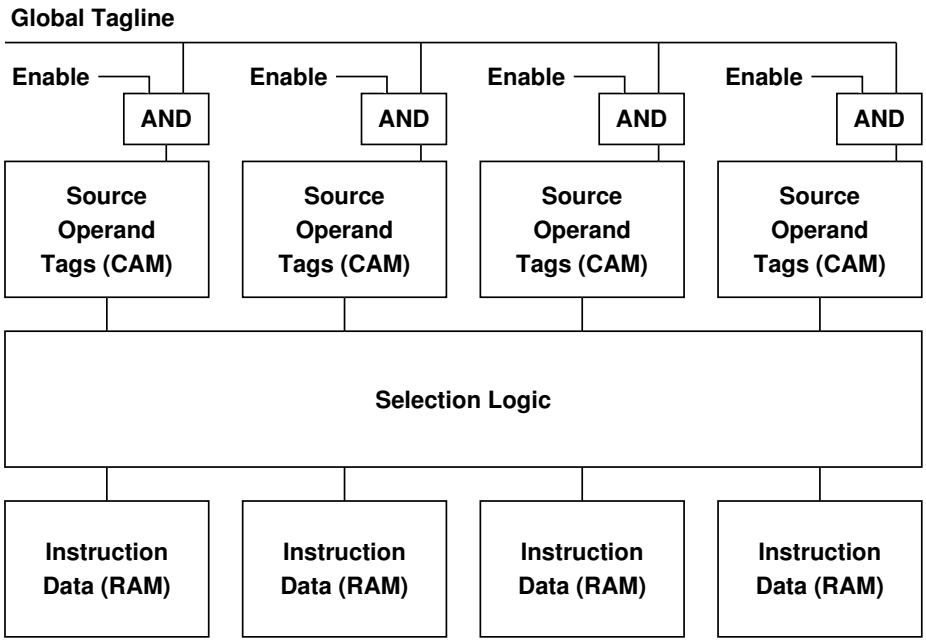


Figure 2.3. A banked issue queue. The source operand tags are stored in expensive CAM cells which provide matching capabilities during instruction wakeup. Other instruction data is held in cheaper RAM cells.

2.2.5 Sources of Energy Consumption

The most energy expensive part of the issue queue is the wakeup logic. Here, CAM cells are used to determine whether a result tag being broadcast to the whole queue matches either of the source operand tags held by the waiting instructions. An example of the wakeup logic for a single bit is shown in figure 2.4 where, for simplicity, the logic to write values into the cell has been omitted.

Each cycle the matchlines are precharged with a high voltage. Tags are broadcast on the taglines and the pull-down stacks used to take the relevant matchline back to a low voltage if there is a mis-match. Each matchline spans the width of the tag, so a mis-match in any bit position pulls it back to a low voltage. An *OR* is performed with all the matchlines to determine if there is a match with any tag and to set the *Ready* bit.

A large amount of dynamic energy is consumed with the constant precharging and discharging of the matchlines. Folegnani and González [32] proposed two optimisations to this logic whereby they prevent empty issue queue entries from being woken and likewise with operands marked as ready. The former is achieved by performing an *AND* of the precharge

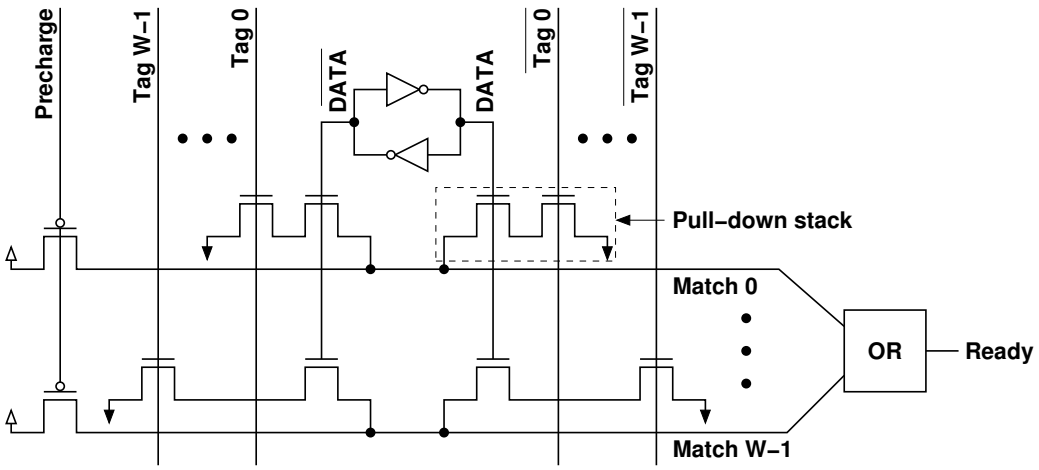


Figure 2.4. Wakeup logic of a CAM cell [66].

signal and the valid bit for each entry before connection to the matchlines. The latter adds a *NOT* of the operand's ready bit to the *AND* so that the entry must be valid and the relevant operand non-ready for the matchlines to be precharged.

In all simulations in this thesis, the baseline simulator has all banks on permanently and does not use the optimisations from Folegnani and González [32]. This is because modern superscalar processors do not implement the issue queue in this way, although they may do in the future, so the baseline microarchitecture is as close to a real processor as possible. However, the issue queue optimisations in chapters 5 and 7 do use these schemes to help save energy.

2.3 Register Renaming

Register renaming is a technique employed to eliminate false (write-after-read (WAR) and write-after-write (WAW)) dependences between instructions and expose greater ILP. It has been used in all modern superscalar processors since the early 1990s [74].

The compiler maintains dependences between instructions by assigning the result value of each instruction to a register. The processor's instruction set architecture (ISA) describes the number of these available, known as logical or architectural registers. It is up to the compiler to assign them as necessary, linking instructions creating a value to those using it. In the portion of a program's control flow graph between a producing instruction creating a value and its last consumer instruction using it, the value is said to be *live*. However, due to the limited number

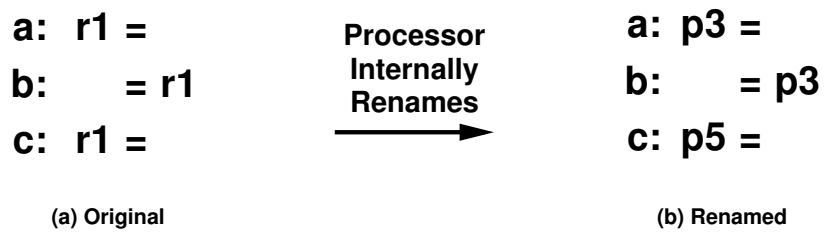


Figure 2.5. Example code the compiler generates and its renamed version.

available, registers have to be reused for the results of different instructions, provided the live ranges of the values do not overlap. This introduces output and anti-dependences between instructions that are actually independent of each other. An output dependence (WAW) occurs when an instruction writes to the same logical register as a predecessor. An anti-dependence (WAR) is an instruction writing to a register that a predecessor reads.

To remove these false dependences between instructions the processor renames the logical registers to unique, temporary storage locations called rename buffers. This creates a one-to-many mapping between logical registers and rename buffers, removing the problem.

For example, consider figure 2.5 which shows two versions of the same piece of pseudo-code. In figure 2.5(a), instruction c is independent of both a and b but there is an output dependence between a and c and an anti-dependence between b and c. If the code were to run on the processor in this form, c would not be able to start execution until a had completed and b started execution too.

Figure 2.5(b) shows the code once it has been renamed within the processor. The result register of instruction a is renamed to p3 and the data dependence between a and b preserved through the renaming of b's source operand to p3 as well. However, the result register of c is renamed to p5 meaning that false dependences on a and b are removed, allowing c to execute entirely independently of the other two instructions.

There are three ways to implement register renaming, each of which provides a different location for the storage of register data. Figure 2.6 shows each type. The first scheme (figure 2.6(a)) uses a register file with separate rename buffers and is described in section 2.3.1. The second method (figure 2.6(b)) is a specialisation of the first, where the rename buffers are combined into the reorder buffer. It is described in section 2.3.2. The final approach (figure 2.6(c)) is to provide a large, centralised register file to store both committed and uncommitted values, which is described in section 2.3.3. Section 2.3.4 discusses two implementations of register

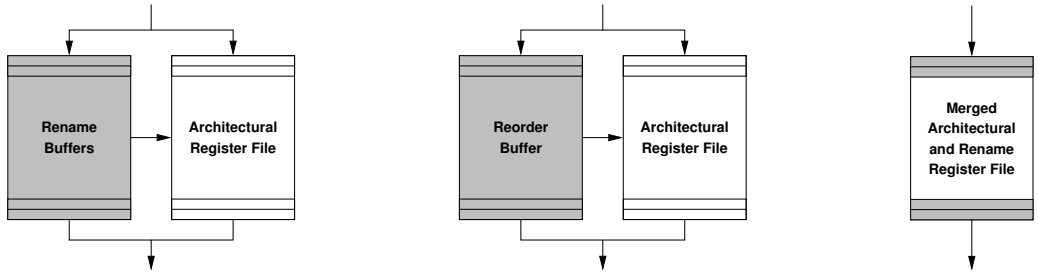


Figure 2.6. Three implementations of register renaming. The shaded area shows the rename buffers [74].

map tables used in register renaming and section 2.3.5 describes the scheme used in this thesis to bank the register file. Finally, section 2.3.6 describes the sources of energy consumption within the register file.

2.3.1 Separate Rename Buffers

The first register renaming implementation has an architectural register file holding committed values for the logical registers, each of which is called a physical register. A separate set of registers is also provided, called the rename buffers, which hold values corresponding to logical registers that have not been committed.

When an instruction dispatches and its logical registers are renamed, a new rename buffer is allocated from a free-list to hold its result. Consumer instructions obtain a tag from the rename buffer for each of their source registers to store in the issue queue for wakeup, as described in section 2.2.3. Once the producing instruction has finished execution the tag is broadcast and subsequent consumers can read the data from the rename buffer. When the instruction is committed, the value in the rename buffer is copied to the architectural register file and the rename buffer put back on the free-list. From this point, consumer instructions must access the architectural register file to get their source data.

Each rename buffer can be in one of three possible states: *Available*, when it is not allocated as any instruction's destination; *Allocated-youngest*, when it is allocated as an instruction's destination register and is the youngest (newest) mapping of the logical register it represents; and *Allocated-older*, when it is allocated but is not the newest mapping of the logical register. Figure 2.7 shows a state diagram for each rename buffer.

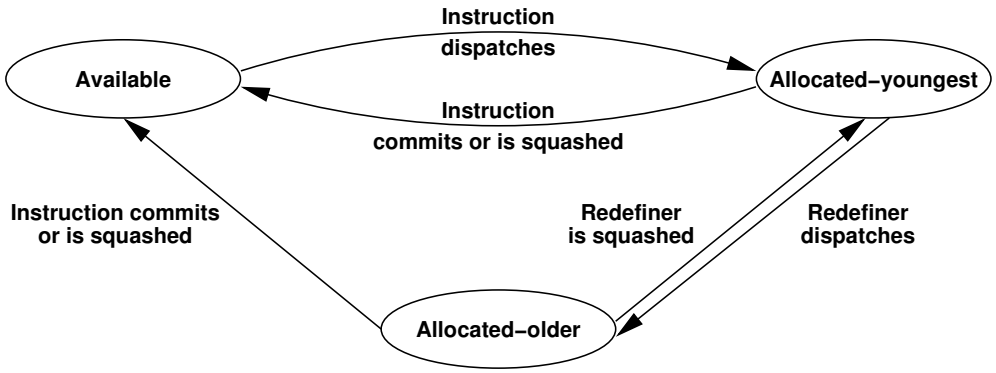


Figure 2.7. The state diagram for a rename buffer [76].

At the start all rename buffers are in the *Available* state. When an instruction that has a destination register R dispatches, a new rename buffer is allocated from the free-list and moves into state *Allocated-youngest*. If the instruction passes through the pipeline without another instruction dispatching that writes to R, the value contained in the rename buffer is moved to the architectural register file when the instruction commits, the rename buffer is put back on the free-list and its state becomes *Available* again. If, however, another instruction writing to R dispatches, called its unique redefiner, the rename buffer moves to state *Allocated-older*. Again, once the first instruction commits the state becomes *Available* after the value has been copied to the architectural register file and the rename buffer put onto the free-list.

When the rename buffer is in *Allocated-youngest* or *Allocated-older* state, its defining instruction can be squashed, so it is put back on the free-list and its state moves to *Available*. If, when in *Allocated-older* state the unique redefiner is squashed, but not the defining instruction, the rename buffer once again becomes the newest mapping of the logical register and is put into *Allocated-youngest* state.

2.3.2 Combined Reorder Buffer

Combining the rename buffers into the reorder buffer is a specialisation of the previous approach. The advantage of this implementation is that there is always a free rename buffer if there is a free reorder buffer entry to dispatch an instruction into. Hence, register renaming never limits the number of instructions dispatching. However, the downsides of this approach are that the reorder buffer is made more complex and there is some inevitable redundancy in-

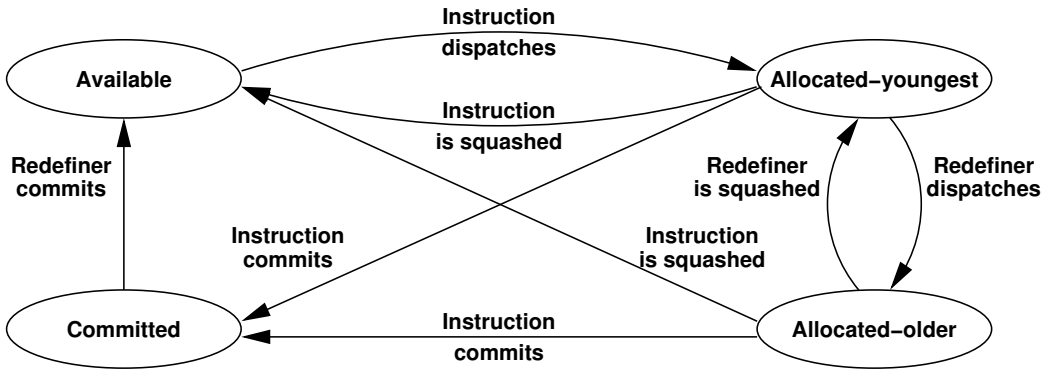


Figure 2.8. The state diagram for a physical register [76].

troduced because stores and some branch instructions do not produce a result, yet are always allocated a rename buffer.

There are examples of processors using separate rename buffers and others using a combined reorder buffer [74]. However, the processor in this thesis uses a different scheme where the rename buffers are incorporated into the register file.

2.3.3 Centralised Register File

The third implementation of register renaming is to store all values, committed or not, in one register file, although there will usually be separate register files for integer and floating point data. In this scheme the rename buffers are called physical registers because they are the same storage locations as the registers that correspond to the committed logical registers.

In this case, the committed value of a logical register is not always found in a particular physical register: its location dynamically changes as the program executes and instructions are committed. A table, called the register retirement map table, records the exact location (physical register) that contains the committed version of each logical register.

Each physical register can, therefore, be in one of four possible states: *Available*, *Allocated-youngest* and *Allocated-older* are the same as with rename buffers (section 2.3.1); and *Committed*, when the data contained in the physical register represents a committed logical register. Figure 2.8 shows a state diagram for each physical register.

At the start all registers are in the *Available* state, apart from those that represent the logical registers, which are in the *Committed* state. There are as many registers in the *Committed* state

as there are logical registers.

Transitions between states occur in the same way as for rename buffers (section 2.3.1), the differences occurring when an instruction commits. In this case no copying is involved but the instruction's destination physical register simply moves into the *Committed* state. It is only when the redefiner commits that the register is put onto the free-list and its state transitions into *Available*.

This thesis uses the centralised register file described in this section as is common in many processors, for example the Sparc64. Henceforth in this chapter, explanations will only refer to this implementation of register renaming.

2.3.4 Map Tables

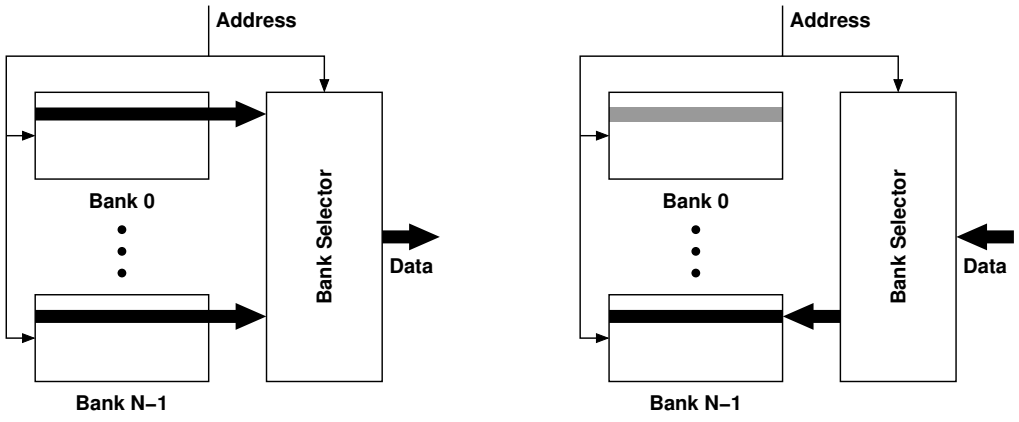
In order that newly dispatching instructions know where their source operands reside (which physical register), a map table is employed, which is consulted and updated when each instruction is renamed. This can either be implemented as an associative lookup using a CAM array, or using an indexed RAM array.

The CAM implementation has as many entries as there are physical registers. An associative lookup is performed in much the same way as a wakeup in the issue queue, described in section 2.2.3, except the tags in this case are the source logical register numbers. To distinguish between multiple mappings, each entry has a *Valid* bit, set when the register is allocated as a destination of an instruction, and a *Latest* bit which indicates whether the physical register is the newest mapping of the source register. This scheme makes recovery from a branch misprediction easy, where all that is needed is to invalidate the entries corresponding to physical registers that are squashed. However, CAM cells are power-hungry and this approach does not scale well.

The RAM scheme has as many entries as logical registers, so is smaller and consumes less energy. It is indexed by logical register number and each entry contains the identifier of the newest physical register to contain the data required.

When any kind of control instruction (branch, procedure call, etc) is dispatched, a copy of the map table is taken, called a checkpoint. These are stored for use in the event of a misprediction on a control instruction to enable fast recovery of the processor's register file state after squashing all mis-speculated instructions (those occurring after the control instruction).

To implement a checkpoint the contents of the map table is copied to some backup storage. This means the number of unresolved branches the processor will allow is dictated by



(a) Register Read (b) Register Write

Figure 2.9. Reading and writing in a register file with *N* banks [4].

the size of the backup storage, i.e. the number of checkpoints that can be taken. If the control instruction is predicted correctly the backup is discarded. However, on a mis-prediction, all instructions younger than the control instructions are squashed and the map table is restored from the backup. In this way the correct mapping of logical to physical registers is maintained. Hence, although the RAM scheme is less power-hungry and scales better than the CAM approach, it requires backup storage for the map table checkpoints for each branch dispatched.

2.3.5 Banked Register File

The register file can be organised into independent banks, in much the same way as the issue queue (section 2.2.4), which can decrease the access time and allow energy savings when turned off.

The register file banking scheme used in this thesis is based on that proposed by Abella and González [4]. To reduce the access time, bank selection and decoding of the entry in the bank are done in parallel. Although this increases the energy consumption compared with a sequential scheme, reads and writes can be performed 27% and 29% faster respectively [4].

Figure 2.9(a) shows the scheme for a register read. The high order address bits are used to select the bank whilst the low order bits choose the entry within the bank. Bank selection is overlapped with the entry decoding and actual reading, so one entry within each bank is read and the bank selection logic chooses the correct data.

For a write, shown in figure 2.9(b), bank selection and entry decoding are performed in parallel. In this situation, an entry is selected in each bank but the data to be written is only sent to the correct bank. Hence, wordlines and bitlines in the other banks are not enabled.

Each bank in the register file can be turned off for dynamic and static energy savings when none of its registers are in use. Dynamic energy savings come about due to the fact that, on a read or write, fewer banks are accessed in parallel. This is achieved by adding a *Busy* bit to each physical register which is set when the register is taken off the free-list and allocated to an instruction. It is reset when freed by the commit of the redefining instruction.

Registers are ordered on the free list by their physical register number. This means that low numbered registers are picked first as destinations for newly dispatching instructions. This concentrates the activity in the low numbered banks, increasing opportunities for turning off the high numbered ones.

2.3.6 Sources of Energy Consumption

Reads consume the most energy in the register file due to the parallel decoding scheme used to enable fast access and the need to drive the data out of the register file. Writes consume slightly less because there is no need to drive data out of the register file. The actual amount of energy consumed in the register file each cycle is dependent on the number of reads and writes occurring and the number of banks actually on. For more information on the energy breakdown of register file operations see chapter 4. As described for the issue queue in section 2.2.5, in the baseline processor all register file banks are permanently on.

2.4 Summary

This chapter has provided background to the thesis, describing a superscalar processor's pipeline with emphasis on the issue logic and register files. It has described various implementations of the issue queue, the banking scheme used in this thesis and the sources of energy consumption, particularly in instruction wakeup. It has described register renaming, a scheme used commonly to eliminate false dependences between instructions. Finally, it has discussed implementations of the rename buffers and explained the register file banking scheme used in this thesis.

Chapter 3

Related Work

This chapter describes related work in superscalar architecture. Firstly, issue queue optimisations, primarily those concerned with reducing the amount of energy in the issue logic, are presented in section 3.1. Previous work on the register file is then described in section 3.2.

3.1 Issue Queue

This section presents work related to the issue queue. There have been many schemes that attempt to remove the need for a monolithic instruction window, from which instructions are sent to the functional units. This is because the wakeups performed in the CAM cells to pass results between producers and consumers are costly in terms of energy and area (see chapter 2), and because the issue queue does not scale well as the number of in-flight instructions increases. For general surveys of issue queue optimisations, see Sima [75] and Abella *et al.* [3].

This section is structured as follows. Section 3.1.1 describes hardware schemes to reduce the number of wakeups in this issue queue. Section 3.1.2 describes approaches that gate off pipeline stages for energy savings. The expected latency of each instruction before issue is calculated by schemes presented in section 3.1.3 whereas section 3.1.4 presents techniques to track dependence chains through the issue queue. Finally, section 3.1.5 describes previously proposed compiler-based schemes.

3.1.1 Reducing Wakeups

This section describes schemes that use the conventional, monolithic issue queue design but attempt to reduce the number of wakeups that occur. They do this by removing some of the CAM cells altogether, gating off wakeups or using improved circuit-level schemes.

3.1.1.1 Folegnani and González (2000 and 2001)

Folegnani and González remove redundant issue queue wakeups by gating off the precharge signal for tag comparisons to empty and ready entries [32]. An AND gate connected to each valid bit, ready bit and the precharge signal achieves this. In chapter 5 the compiler-directed scheme for saving energy in the issue queue makes use of this approach to avoid waking empty operands.

The authors also propose a scheme to dynamically resize the issue queue, increasing the number of empty entries, thus reducing the number of wakeups [31, 32]. They monitor the number of instructions issued from the youngest part of the queue over a period of 1000 cycles and the number of entries allowed to contain instructions is adjusted accordingly. Their results show an average reduction of 90.7% in the wakeup logic energy with a 1.7% performance loss. However, using the recent program history to determine future issue queue size requirements is not always accurate and rapid phase changes cannot be accommodated when adjustments are made infrequently.

3.1.1.2 Huang *et al.* (2002)

Huang *et al*. [42] build on the work of Folegnani and González [32] by eliminating wakeups of most valid, non-ready operands. Producer instructions are connected to consumers via extra fields in the register rename map table and reorder buffer. A producer instruction writing back to only one consumer simply forwards the data rather than performing a global wakeup. Results show that tag comparisons can be reduced by 99% with no drop in performance. However, this scheme greatly increases the complexity of the issue queue and dispatch logic, whilst only providing modest additional benefits to Folegnani and González.

3.1.1.3 Ernst and Austin (2002)

Ernst and Austin propose an issue queue that stores only one source operand tag per instruction [28]. When an instruction dispatches with multiple unready source operands, a predictor

is employed to estimate the latest operand to become ready. This approach experiences a slight reduction in IPC but creates savings of 30% to 60% in the scheduling logic energy. The downside to this scheme, however, is a reduction in the flexibility of the architecture. This issue queue structure would not perform well on programs that contain many instructions with multiple source operands, or on those where it is difficult to predict the latest operand to become ready.

3.1.1.4 Buyuktosunoglu *et al.* (2002)

Buyuktosunoglu *et al.* propose a banked issue queue that only stores one source tag per instruction [19]. A small, conventional issue queue is provided for instructions with multiple unready source operands at dispatch. Again, this scheme would not perform well when a program contains many instructions with more than one source operand because the small, conventional issue queue would fill quickly leading to frequent dispatch stalls.

3.1.1.5 Critical Analysis

The designs presented in this section have focused on reducing the number of expensive wakeups occurring when results are broadcast to the issue queue. Some schemes simply gate-off the precharge signal to certain tags (those for ready, empty and some non-ready operands) which saves energy at wakeup without sacrificing flexibility [31, 32, 42]. However, these approaches can increase the complexity of the queue and dispatch logic as consumer instructions are linked to their producers.

Other approaches alter the layout of the queue, providing some entries with zero, or only one tag [28, 19]. However, the result of these changes is a loss of flexibility and consequently this is compensated by the addition of a predictor or a small, conventional queue. The schemes presented in the next section attempt to retain the current architectural flexibility but save energy by turning off under-used parts of the processor when the effect on performance would not be too severe.

3.1.2 Gating Pipeline Stages

When there is low instruction level parallelism (ILP) available in the issue queue, more instructions get dispatched than are executed, causing the queue to fill up. Energy is then wasted waking instructions whilst functional units wait idly. This section describes approaches that gate instruction fetch, dispatch or issue or turn off parts of the pipeline for energy savings.

3.1.2.1 Manne *et al.* (1998)

Manne *et al*. gate off parts of the pipeline when too many speculative branches have entered it [56]. Each branch is estimated as being high-confidence, meaning it is likely to have been predicted correctly, or low-confidence, meaning the opposite. When there are too many low confidence branches in the pipeline then fetch is gated until some have been resolved. Results show a 38% reduction in wrong-path instructions with a performance loss of 1%. However, incorporating the confidence estimators into the branch predictor could allow for better branch prediction and better overall performance.

3.1.2.2 Maro *et al.* (2000)

Maro *et al*. save power in their processor by putting it into one of four low power modes [57]. These correspond to different configurations of the two integer and one floating point clusters of functional units., one integer cluster always being on. Issue IPC and data dependences in the reorder buffer are monitored to provide conditions for mode transitioning. On average, 1% performance loss is experienced with a 10% reduction in energy consumed. However, this provides a course-grained approach to energy reduction where more savings could result from a scheme that adjusts rapidly to changing program requirements.

3.1.2.3 Bahar and Manne (2001)

Bahar and Manne reduce the issue width of the processor from 8-wide to either 6-wide or 4-wide depending on the issue IPC over a fixed number of cycles [11]. Results show a performance loss of 1% with 10 - 23% issue queue energy savings and a 5 - 12% reduction in functional unit energy. Again, a fine-grained scheme could provide further energy savings throughout the processor. In addition to this, it uses recent program characteristics to determine future requirements, which may not always correctly represent the next program region.

3.1.2.4 Buyuktosunoglu *et al.* (2000, 2001 and 2003)

Buyuktosunoglu *et al*. propose a banked issue queue design [21, 18]. Each bank can be independently turned off for energy savings. The issue queue design is that used in this thesis and is described in detail in chapter 2.

The committed IPC and occupancy of the queue are measured over a fixed number of cycles and used to determine the number of banks to disable for the next sampling period. Results

show a performance decrease of 3.1% with a 31.7% saving on energy within the issue queue and associated logic with one set of thresholds, or 18.2% energy savings and 1.0% performance loss with another. In a further work [20], fetch gating is combined with this issue queue size adaption for increased energy savings.

This scheme, once again, uses course-grained analysis of the recent program region to guide energy saving decisions for the future. Greater savings could result from an exact, fine-grained knowledge of the future to accurately fit resources to the program's requirements.

3.1.2.5 Abella and González (2002 and 2003)

The issue queue and reorder buffer are dynamically resized by Abella and González to reduce the amount of energy they use [1, 5]. This is performed in a manner similar to Folegnani and González [32], whose scheme to gate off wakeups to empty entries is used. The issue queue structure is similar to that proposed by Buyuktosunoglu *et al*. [21] where the queue is banked and energy savings resulting from turning off empty banks.

The authors dynamically alter the size of the reorder buffer and issue queue after gathering statistics over a number of cycles about the occupancy of both structures. Compared with the scheme proposed by Folegnani and González [32], this approach incurs a smaller performance loss and saves more energy in the issue queue and register files. However, the downside is that the course-grained, recent program region analysis does not save as much energy as a fine-grained approach which could also incur a smaller performance loss.

3.1.2.6 Critical Analysis

This section has presented techniques to dynamically fit processor resources to the program requirements. Some schemes gate off parts of the pipeline when the IPC is below a sufficient threshold, thereby saving energy in many stages. Others focus on the issue queue and use a combination of IPC and queue occupancy to guide their dynamic resizing decisions. However, these schemes employ course-grained analysis, only taking energy saving decisions after a fixed sampling period or when a certain condition is reached. In addition to this, the requirements of the most recent program region are used to guide decision making for the future, which may not be accurate. Schemes presented in the next section remove the wakeups completely, instead of limiting the number that occur. This is performed by calculating the time an instruction must wait before it can issue and placing it in a structure that allows it to issue once this time has expired.

3.1.3 Latency Calculation

Knowing the issue time of previous instructions means that the issue time of newly dispatching instructions can be calculated and used to avoid wakeups within the issue queue. This value, the number of cycles between dispatch and issue, is referred to as the issue latency. Schemes that attempt to use this information are presented in this section.

3.1.3.1 Canal and González (2000 and 2001)

Canal and González present a scheme which records the cycle each physical register is expected to be valid [22]. The issue queue is modified so that instructions can only issue from the head, which moves back each cycle. Instructions are placed in the queue so that all input dependences are satisfied once they reach the head, with a small, conventional queue provided for instructions dependent on loads. In a second paper, the authors extend their proposal by assuming that loads will hit in the L1 cache, thereby removing the need for this extra queue [23].

The advantage of these schemes is that there are few expensive wakeups because no results are broadcast to the main issue queue. However, the energy savings are not quantified and experiments show a performance loss of about 4% compared with a machine with a conventional issue queue

3.1.3.2 Ernst *et al.* (2003)

Ernst *et al*. propose a new wakeup-free issue queue layout with a prescheduler to determine the number of cycles each instruction must wait before being issued [29]. A new queue is provided into which instructions dispatch and wait for a number of cycles. They are then moved to a second queue to be issued.

Although this scheme removes the wakeups from the issue queue, the benefits are not quantified. Performance compared to a traditional microarchitecture is consistently less, especially in benchmarks experiencing high cache misses. This is because many instructions reach the head of the second queue without having all source operands ready and must therefore be rescheduled and move through both queues again.

3.1.3.3 Hu *et al.* (2004)

Hu *et al*. propose a segmented issue queue where instructions are issued from the lowest segment only [40]. At dispatch, the issue latency is calculated and instructions dispatched to an issue queue segment accordingly. They then move towards the lowest segment after waiting a number of cycles in each intervening one.

Using this scheme, performance is reduced by 5.8% over the baseline with a conventional issue queue. The energy savings due to a lack of wakeups are not quantified. In addition to this, load predictors are needed to help estimate the latency of each dispatched load and these may not always be accurate.

3.1.3.4 Raasch *et al.* (2002)

A segmented issue queue without wakeups is also proposed by Raasch *et al*. [70]. They link dependent instructions together into chains using loads as the head of each new chain. The issue latency of each head is calculated and the values for the other instructions in the chain is determined relative to this. Instructions move between segments depending on their current issue latency and can issue from the lowest segment only.

Results show performance losses of 2% to 45% over a baseline processor with a similar sized, conventional issue queue. One disadvantage of this scheme is that the number of in-flight chains allowed in the issue queue must be hardwired. Programs with a large number of small chains may perform badly on this architecture.

3.1.3.5 Michaud and Seznec (2001)

Instruction prescheduling is proposed by Michaud and Seznec to provide a large instruction window and short clock cycle [60]. An extra pipeline stage is provided after dispatch, where instructions are placed into a preschedule buffer at the same time as their expected issue cycle is calculated. A small, conventional issue queue is provided to issue instructions and they are moved to it depending on their issue latency.

This scheme is not compared to a baseline, therefore performance results are difficult to analyse. Furthermore, the introduction of an extra pipeline stage increases complexity and the branch mis-prediction cost. In addition to this, benefits from reduced issue queue size are not quantified meaning that it is difficult to determine the advantages of this approach compared to other latency-based schemes.

3.1.3.6 Critical Analysis

This section has considered schemes that determine each newly dispatching instruction's latency before it issues to provide microarchitectures with reduced or completely eliminated wakeups. The downside to these schemes is that they are often dependent on predictors to decide whether a load is going to hit or miss in the L1 cache. When a miss does occur, it is not just the immediate dependents that need to be updated, but the whole dependence chain. This increases the complexity of the new hardware or wastes bandwidth as instructions continue to move through the queue structures with the wrong latency and are squashed when they come to issue. To alleviate this problem, the schemes presented in the next section track dependence chains as they pass through the pipeline so that any changes to an instruction's issue time can be easily propagated to its dependences.

3.1.4 Dependence Tracking

Dependences between instructions are easily recognised based on the reading and writing of registers. This section presents schemes that provide structures enabling the forwarding of data through these dependences, removing the need for costly wakeup logic.

3.1.4.1 Canal and González (2000 and 2001)

Canal and González implement the issue queue as a table with one entry per physical register [22]. Instructions that are the first consumer of their non-ready source registers are placed in this first-use table, those with all operands ready are placed in a ready queue for issuing in order and all others are placed in a small, conventional issue queue. In a second paper, the first-use table is replaced by an N-use table [23].

Experiments show that a 2-use table with an issue buffer of 2 entries gives a slowdown of about 4% over an out-of-order issue machine. This approach only works well when there are many 1-use or 2-use values present in the program. If there are many operands that are used more than this then the size of the issue buffer is critical and greater performance losses could result.

3.1.4.2 Önder and Gupta (1998)

A new superscalar architecture is proposed by Önder and Gupta in which the issue queue is split and each part holds pointers of dependences between operands [65]. These are placed next

to the functional units and when an instruction has one or more of its source operands unready, a link is made from the producer's entry to the consumer's, enabling data forwarding.

There is a small performance loss using this scheme compared with a baseline processor with a traditional issue queue. In addition to this, the benefits of the new architecture, in terms of energy savings, are not detailed. Furthermore, flexibility is lost by splitting the issue queue into separate parts.

3.1.4.3 Palacharla *et al.* (1997 and 1998)

A generic superscalar processor pipeline is defined by Palacharla *et al*. and the complexity of various components analysed [67, 66]. The rename, wakeup, selection and data bypass logics are described and analysed in depth and a delay analysis performed which is verified in Hspice.

The authors replace the issue queue with first-in-first-out (FIFO) queues which can only issue in-order. Consumer instructions are put in the same queue as their producer, if possible, or in an empty FIFO. A complexity analysis of the new design is made and it is found that the performance is comparable to the baseline. However, there is a loss of flexibility and programs with a large number of short dependence chains may experience high performance losses due to a lack of empty FIFOs into which instructions can dispatch.

3.1.4.4 Abella and González (2004)

Abella and González analyse the FIFO issue queue structure proposed by Palacharla *et al*. [67] but find that it performs badly for floating point codes [6]. Their proposed approach replaces the FIFOs with RAM buffers, each with a number of identifiers associated with it, which represent dependence chains. Newly dispatching instructions are placed in the buffer that contains their producer instruction, but only if the producer is the last instruction in its dependence chain. Otherwise a new chain identifier is used (possibly from a different buffer) and the instruction dispatched accordingly.

Results from the new scheme show further reductions in IPC loss for floating point benchmarks with large energy savings over the baseline processor with a conventional issue queue. This scheme still could suffer a performance loss when there are many short dependence chains unless the number of chain identifiers is large.

3.1.4.5 Critical Analysis

Schemes that track dependence chains as they pass through the issue logic have been considered in this section. One scheme has included a table indexed by physical register number to directly link a number of consumers to their producer [22, 23]. However, a small, conventional issue queue must still be provided when there are more consumers than can be included in the table, and for many registers this table could quickly become large.

Other approaches have considered using FIFOs or RAM buffers to hold instructions, removing the need for expensive wakeup logic [67, 6]. However, flexibility is lost which could result in performance losses when there are a large number of short dependence chains within the program.

Finally, one further technique has attempted to easily link producers to consumers [65]. However, flexibility is again lost due to the implementation of the issue queues holding instructions that are waiting to issue and those executing.

3.1.5 Compiler Schemes

Most compiler approaches to energy reduction focus on very long instruction word (VLIW) machines where the compiler has, typically, more control over instruction scheduling. This section presents a variety of work based on compilation for both VLIW processors and dynamic voltage scaling in superscalars.

3.1.5.1 Toburen *et al.* (1998)

Toburen *et al*. propose an instruction scheduling algorithm for a VLIW processor [81]. Based on the traditional list scheduling algorithm, each cycle only a limited number of the available instructions are actually scheduled. The energy that any instruction will dissipate when it executes on a given functional unit is known and so the total energy dissipation for any cycle is limited. Hence, once the energy threshold for the current cycle being scheduled is reached, no more instructions are scheduled and the next cycle's scheduling starts. The downside to this approach is an unavoidable performance loss if the energy threshold is too low, or few energy savings if it is too high.

3.1.5.2 Lorenz *et al.* (2001)

Lorenz *et al*. use a phase-coupled, genetic algorithm to produce code which minimises memory accesses and schedules instructions for energy efficiency [53]. They obtain, through simulation, the energy dissipated by certain combinations of instructions and use these values to predict the energy cost for a sequence of instructions. Results show memory access reductions of 18% to 58% and overall energy reductions of 18% to 36% when compared to a tree-based scheduling approach. However, this scheme is targeted to an in-order architecture where the compiler schedules the code. In an out-of-order superscalar processor, this approach is inappropriate because the compiler cannot know the exact order in which instructions will be executed.

3.1.5.3 Zhang *et al.* (2001)

Zhang *et al*. use the slacks occurring in a VLIW schedule to produce code that reduces a program's dynamic and static energy requirements [86]. Several versions of each functional unit are provided, each operating at a different voltage, which can be individually gated off to save static energy when not in use. When an instruction has some slack behind it in the schedule, a low voltage functional unit can be used to execute slowly, but with a low dynamic energy consumption. However, for this scheme to be useful, several versions of each functional unit must be available, increasing the overall area requirements and adding to the complexity.

3.1.5.4 Lee *et al.* (2000)

Lee *et al*. reduce the switching activity on a VLIW instruction bus by rearranging the already-scheduled micro-instructions [50]. The scheme minimises the hamming distance between consecutive micro-instructions going to the same functional unit. All optimisations are performed at a basic block level. Results show the average reduction in switching activity is 13.30%. However, this is, again, only applicable to in-order processors.

3.1.5.5 Hsu *et al.* (2000 and 2001)

Hsu *et al*. describe a compiler technique to dynamically scale the processor's operating voltage in different program regions [38, 39]. They define conditions based on the percentage of program execution time that the processor alone is busy, the memory system alone is busy, or both are busy. They allow a certain amount of performance loss and determine a slowdown factor, and hence a new voltage, that achieves this loss.

The authors consider the effect of loop-level optimisations on the ability of the compiler to reduce the voltage. They also describe an algorithm to select program regions suitable for dynamic voltage scaling. Results show a slowdown of 2.5% can bring energy savings of between 4% and 24% for floating point programs. The downside of these schemes being the performance loss incurred for potentially small energy savings.

3.1.5.6 Magklis *et al.* (2003)

Magklis *et al*. use the compiler to apply dynamic voltage scaling instructions to programs that have been profiled on training data to save energy in a multiple clock domain processor [55]. The authors propose a four stage algorithm to produce programs which dynamically alter the voltage and frequency of each domain. The algorithm uses profile information and works towards a user-defined performance loss. Results show 31% energy savings with a 7% slowdown which is better than a hardware-determined voltage scaling algorithm. It is also close to an oracle scheme that has perfect future knowledge, although the performance loss is high.

3.1.5.7 Critical Analysis

The schemes presented in this section target both VLIW and superscalar processors. Unfortunately, those techniques aimed at the embedded domain that rely on altering the instruction schedule cannot be transferred to an out-of-order superscalar because the compiler does not know the exact order of instruction execution. Some schemes that perform dynamic voltage scaling experience high performance losses with few energy savings.

3.2 Register File

This section describes work related to optimising the register file. Some schemes have been implemented for comparison later in this thesis with the approaches described in chapter 6. These are indicated and described in detail where necessary.

Moudgill *et al*. proposed the unified register file where registers that hold speculative values are in the same storage area as those that contain committed values [64]. They describe schemes to reclaim each physical register using an in-order map table and freeing the previous version of the logical register once each instruction completes.

Abella and González evaluate their issue queue limiting scheme on the register file and rename buffers to reduce register pressure [4]. For more information on this scheme see section 3.1.2.5 and for general overviews of register file designs see Sima [76] and Zyuban and Kogge [88].

This section is structured as follows. Section 3.2.1 describes schemes to optimise the register file based on the values it contains. Section 3.2.2 presents techniques to allocate register late and release them early. The number of ports to the register file are reduced by schemes presented in section 3.2.3 and section 3.2.4 describes methods of register caching. Finally, compiler-based schemes are presented in section 3.2.5.

3.2.1 Value Optimisation

The redundancy present in the values stored in the register file is optimised by schemes described in this section. The redundancy occurs because many of the stored values are just a 1 or have all bits 0 and do not need the whole 64 bits provided by each register. Many values are also duplicated elsewhere in the register file.

3.2.1.1 González *et al.* (2004)

Two values are similar, as defined by González *et al.*, if they are microarchitecturally near to each other: differing in their least significant bits only, a description which captures partial value locality [35]. The authors propose a new integer register file organisation consisting of three structures. One holds 64-bit data whilst another stores only high-order bits for values that are present in the third register file. On a read, an extra pipeline stage is added to allow for sequential access when the required value is in two of the three structures. Writes also must take an extra cycle to detect the size of the value being written and to store the data in the correct structures.

Results show a 50% energy saving with an IPC loss of 1.7%. The access time reduction is up to 15% which could allow a faster clock frequency. However, the addition of extra pipeline stages when reading and writing values increases the complexity of the bypass logic and increases the branch mis-prediction penalty.

3.2.1.2 Kondo and Nakamura (2005)

Kondo and Nakamura propose the bit-partitioned register file (BPRF) which, instead of providing one bank of 64-bit registers, instead provides two banks of 32-bit registers [47]. For a 64-bit

value, one register in each bank is allocated where one provides the most significant bits and the other provides the least significant bits. This scheme achieves an increase in performance, especially for small register files, although an increase in energy consumption can occur too.

3.2.1.3 Ergin *et al.* (2004)

Register packing, allowing several values to be held in one physical register, is proposed by Ergin *et al*. [26]. A 4-bit mask is included with each physical register tag to indicate the 16-bit fields within the register that contain the required data. In addition, a free list is maintained for each 16-bit data size multiple available.

The author propose speculative packing where predictors estimate the width of the result before execution and assign a destination register accordingly. This brings a 15% improvement in IPC. However, a deadlock avoidance strategy is required and extra tags have to be broadcast if a mis-prediction occurs.

3.2.1.4 Lipasti *et al.* (2004)

Lipasti *et al*. propose storing narrow register values in the register map table when a RAM implementation is used in register renaming [51]. The map table is indexed by logical register number and each entry contains a pointer to the physical register holding the latest value. The authors propose storing either the pointer or the narrow value with an extra bit indicating the addressing mode (i.e. whether it is an immediate value or a pointer to the physical register).

Results show performance improvements of 15% due to reductions in register pressure. However, the map table and backup copies (kept for branch mis-prediction recovery) are made more complex. This is because they must be writable from the commit stage.

3.2.1.5 Balakrishnan and Sohi (2003)

Balakrishnan and Sohi present three schemes to exploit duplication in the register file [12]. The first maps many logical registers to one physical register when the values are the same. The second avoids duplicating the values 0 and 1 by providing dedicated tags representing these values. The third approach extends the second by removing the need for re-broadcasting the tags to the issue queue when a 1 or 0 is produced. However, to implement these schemes a register cache must be implemented and bits added to each register, increasing the complexity of the microarchitecture.

3.2.1.6 Critical Analysis

This section has presented schemes that take advantage of similar values within the register file. Some approaches propose different layouts of the register file to implement their technique, whereas others modify the existing structures. In most schemes, the complexity of the register file or writeback logic is increased to recognise and optimise similar values. This could mean the addition of an extra pipeline stage which increases the branch mis-prediction penalty.

In general, these approaches rely on the characteristics of programs being executed. As such, programs with few duplicated or narrow-width operands will not perform well. The schemes presented in the next section are orthogonal to these and do not rely on program characteristics such as these. Instead, they attempt to reduce the idle time a register experiences either before it is written or after its last read.

3.2.2 Late Allocation and Early Release

This thesis presents a number of compiler-directed early register releasing schemes in chapter 6. These are motivated by the fact that many registers are idle for a number of cycles between the execution of their last consumer instruction and the commit of the instruction redefining their logical register. Early releasing attempts to release registers during this idle time, as soon after the execution of the last consumers as possible.

Registers are also idle for many cycles before the producer instructions finish execution and write to them. Late allocation schemes attempt to reduce this idle time whilst still maintaining data dependences and eliminating false dependences. This section describes previous approaches to both problems.

3.2.2.1 Wallace and Bagherzadeh (1996)

Wallace and Bagherzadeh split their register file into several banks each with only one write port and two read ports [85]. During register renaming, instructions are given a unique tag for their results but a register is not actually allocated until execution is complete. If possible, the result is written over the previous version of the logical register, reducing register pressure. The downside to this scheme is that deadlock occurs if the oldest instruction within the pipeline does not have a register to write to after execution. Strategies must be put in place to avoid such a situation.

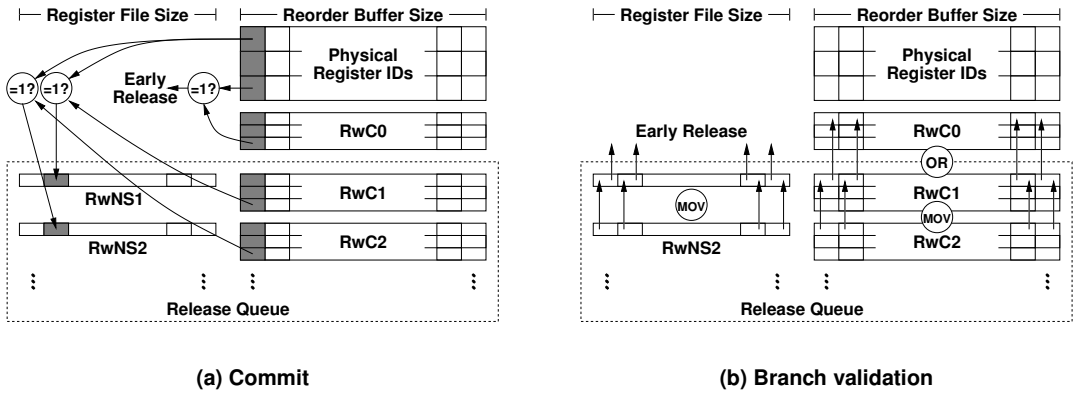


Figure 3.1. Early releasing as proposed by Monreal *et al.* [62].

3.2.2.2 González *et al.* (1998) and Monreal *et al.* (1999)

Physical registers are allocated late in the pipeline by González *et al*. through the use of virtual-physical registers [34]. These are unique tags which are assigned to an instruction's source and destination operands at dispatch instead of the physical register tag. A physical register is not mapped to an instruction's destination operand until execution is complete, reducing register pressure.

The scheme requires several additional map tables and the ability to broadcast both virtual-physical and physical register tags to the issue queue. Deadlock can occur if no physical registers are available to be allocated as a destination for the oldest instruction in the reorder buffer. A scheme is presented to prevent this and is improved upon in a later paper by Monreal *et al*. [61].

Results show a 19% increase in IPC or a 26% reduction in register pressure. However, the need to pass the virtual-physical register tags around increases the processor's energy consumption and care has to be taken to avoid deadlocks if there are few physical registers.

3.2.2.3 Monreal *et al.* (2002 and 2004)

Monreal *et al*. propose hardware schemes to allow the early release of registers [62]. These work by identifying the instruction that is the last use of each logical register and the instruction redefining it. A release queue is implemented to hold this information and determine when a register can be released. This is the latter of the last use instruction committing and all branches before the redefiner being confirmed as non-speculative.

Figure 3.1 shows how early releasing takes place in this scheme. The release queue is com-

posed of two parts: a release when commit (RwC) structure and a release when non-speculative (RwNS) structure. These hold information about the last use and redefining instructions as well as the registers to release when the oldest branch is confirmed.

Results show speedups of 5% for integer and 8% with floating point codes with a small register file. However, if an exception were to occur between the early release of a register and the commit of its redefining instruction, a precise processor state would not be recoverable.

In a further paper [63], the authors combine this with work on virtual-physical registers [61] to reduce physical register idle time and register pressure both before a value is written to a register and after the commit of the last user of that value. Although this can bring performance increases of 15% for small register files, the new scheme still does not implement precise exceptions.

3.2.2.4 Ergin *et al.* (2004)

The checkpointed register file is proposed by Ergin *et al*. to help with the implementation of early register releasing [27]. This can be used to store a backup of a register when it is released early so that its value can be recovered in the event of a branch mis-prediction.

A register is released early when its defining instruction commits or when its redefining instruction is renamed. In both cases, all consumers of the register must have started execution. Results show a performance increase of 23% for a small register file. However, the downside of this scheme is that a register cannot be released before the redefining instruction enters the pipeline, when it may have been idle for many cycles before this.

3.2.2.5 Martínez *et al.* (2002)

The Cherry scheme, proposed by Martínez *et al*., attempts to recycle all instruction resources early, rather than just the registers [59]. Load queue entries are released as soon as the relevant load has executed, all previous loads have been released and older stores' addresses are known. Store queue entries are recycled in a similar fashion except that stores cannot be speculative and all older loads' and stores' addresses must be known. The cache is augmented with extra bits to indicate speculative stores. To release registers early, the redefining instruction must be non-speculative and all consumers must have executed.

Results show a 6% increase in performance for a processor with 192 integer registers, a reorder buffer of 384 entries and 32 entry load and store queues. However, this scheme requires

major changes to the microarchitecture and it is necessary to periodically take backups of the processor state in case an exception or interrupt occurs.

3.2.2.6 Akkary *et al.* (2003)

Akkary *et al*. present an analysis of the requirements of various structures in the processor pipeline when allowing a large number of in-flight instructions [9]. They propose changes to the register map table and store queue to reduce complexity. Physical registers are reclaimed early by counting the consumer instructions and monitoring the redefining instruction, as proposed by Moudgill *et al*. [64].

Performance results for the processor created when combining all schemes show that the new design performs better than a conventional reorder buffer based processor with the same cycle-critical buffer sizes. However, as with the Cherry scheme [59], this approach must take periodic backups of the processor state, requiring large, additional structures to hold the saved values.

3.2.2.7 Critical Analysis

The schemes presented in this section have considered late allocation and early release of processor resources. By using virtual-physical registers [34], registers can be allocated later than usual. However, deadlock avoidance strategies must be designed to ensure the oldest instruction in the pipeline can always make progress.

Early releasing processor resources can reduce register pressure effectively, providing performance gains. However, to implement precise exceptions, periodic backups must be taken of the processor state, requiring additional, expensive storage to hold the saved values and increasing the microarchitectural complexity. The schemes presented in the next section do not require these expensive backups, instead reducing the port requirements of the register file.

3.2.3 Ports Reduction

Increasing the physical size of the register file increases its access time. Apart from increasing the number of registers, increasing the number of ports to the register file can also enlarge it greatly too. In fact, doubling the number of ports has a squaring effect on the area of the register file because the number of wordlines and bitlines must both be doubled.

This section looks at work performed in optimising the number of ports in the register file, motivated by the fact that the bypass network supplies many operands to issuing instructions.

3.2.3.1 Park *et al.* (2002)

Park *et al*. add a bypass hint bit to each operand in the issue queue which helps determine whether an operand should come from the register file or through the bypass network [68]. The register file is not read if an operand should appear on the bypass network. The authors' register file is implemented with fewer ports than usual due to a decrease in the number of reads. It is also banked for writing and virtual-physical registers [34] are used.

Results show a 9% saving in the energy-delay product but the downside is an extra pipeline stage between issue and register read. In addition to this, an incorrect bypass hint bit incurs a cycle penalty with issue stalled whilst the correct value is read.

3.2.3.2 Tseng and Asanović (2003)

Tseng and Asanović present a design for a register file split into banks with only two read ports each [83]. Each functional unit is given two global read ports, allowing them to be connected to each register file bank. Although results show an access time reduction of 20%, an IPC loss of 5% is incurred. In addition to this, an extra pipeline stage is introduced between the issue and register read stages to deal with arbitration on the read ports. If there is a conflict, instructions not getting a read port are squashed along with all instructions issuing in the previous pipeline stage, introducing a 2 cycle penalty.

3.2.3.3 Aggarwal and Franklin (2003)

Aggarwal and Franklin reduce the width of some of the register file read and write ports [7]. Certain functional units are given narrow read and write ports, restricting the instructions that can be executed on them. Within the register file, a bit for each register is added to determine whether the values held are narrow or not. These are used when issuing to determine which functional unit the consuming instruction can be sent to. Results show power savings of 60% with a 3% IPC loss. However, when executing programs with a large number of wide values, the performance loss could be much higher as the number of available functional units is restricted.

3.2.3.4 Kim *et al.* (2003)

Kim *et al*. reduce the number of read ports on the register file through the addition of a delayed writeback queue [45, 46]. This holds results from instructions for two cycles after writeback,

reducing the need for operands to be read from the register file. An operand prefetch buffer is also introduced which fetches valid operands at dispatch and stores them whilst instructions wait in the issue queue, provided there are unused read ports during the dispatch cycle. Finally, an operand prefetch request queue is proposed to cache requests and perform them when resources become available.

Results show a 1% performance loss and 22% energy savings using these structures and halving the number of register file read ports. However, the addition of these queues and buffers increases the complexity of the dispatch and issue stages and to determine where values reside, extra bits are required in the issue queue. Port contention can also hinder performance when many instructions require operands that have not been prefetched from the register file.

3.2.3.5 Seznec *et al.* (2002)

Seznec *et al*. propose a new architecture for clustered superscalar processors to reduce the register file port requirements [72]. The register file is split into several distinct subsets with each cluster of functional units allowed to write into only one subset. Register renaming is altered such that instructions are first assigned to clusters, then renaming takes place, so as to take into account the physical register subset that the functional unit cluster can write to. This could add extra pipeline stages and deadlock has to be carefully avoided when there are fewer physical registers in a subset than there are logical registers. In addition to this, instructions executed on a particular cluster must read their operands from a fixed pair of register subsets and write their result to a fixed subset. Therefore, each cluster must be able to execute each type of instruction for this architecture to work and the register renaming pipeline is more complex than for a superscalar or conventional clustered processor.

3.2.3.6 Kucuk *et al.* (2002 and 2003)

A reorder buffer with no ports for reading operands is presented by Kucuk *et al*. for use when the reorder buffer holds uncommitted values [49]. When an instruction is committed from the reorder buffer, its result is written into the physical register file and broadcast to the issue queue if another instruction has requested it. Instructions request a re-broadcast if they do not manage to read the value from the bypass network. Retention latches are also added to cache results written to the reorder buffer, from where they can also be read. Results show up to a 3% performance improvement over a 2-cycle reorder buffer with full bypassing. The reorder buffer area can be reduced by 45% and the overall chip power by 6% to 8% The authors then

extend the work in a new paper by preventing short-lived values from being written into the retention latches [48].

Although these schemes can reduce the reorder buffer's energy requirements, they can increase the issue queue's. This is because there may be many values that are broadcast to the queue when their producer writes back and are re-broadcast when it commits. This could also lead to contention for the taglines in the issue queue, increasing the complexity of both issue and commit.

3.2.3.7 Savransky *et al.* (2004)

Savransky *et al*. present a scheme called lazy retirement which only moves results from the reorder buffer to the correct destination register when the reorder buffer entry is going to be overwritten with a newly dispatching instruction [71]. They implement this through a new map table which links each logical register to the position of its last committed value (either in the reorder buffer or the physical register file). Results show 75% of the copies from reorder buffer to the register file are eliminated. However, this scheme does not work for register renaming implementations that store uncommitted results in a centralised register file.

3.2.3.8 Critical Analysis

Schemes that reduce the number of ports to the register file have been presented in this section. Some approaches alter the layout of the register file and only allow values to be written into certain parts of it, others optimise for clusters of processors. In some schemes, the frequency of narrow-width operands is exploited, whereas other techniques involve adding structures to prefetch or cache registers. Most of the schemes presented experience performance losses. The main problem with reducing the number of ports is that reads and writes to the register file cause port contention, delaying certain operations and increasing the complexity of various pipeline stages.

The main motivation for reducing the ports in the register file is that it decreases the area and access time. The schemes in the next section attempt to compensate for a long latency register file, possibly reducing the port requirements as a side effect, by implementing register caching.

3.2.4 Register Caching

The current trend towards multi-threaded processors means that register file sizes need to increase to provide enough resources for each thread. However, as the size increases so does the access time to read or write a value, meaning the register file becomes a bottleneck to performance.

Instead of creating a register file with a multi-cycle access latency, one solution to this problem is to separate the register file into several banks, each containing a subset of the total registers. This reduces the access time because the delay through a single bank is smaller than that through a large, un-banked register file. It is also beneficial in terms of energy because the word and bitlines within each bank are much shorter.

The register cache is one example of banking. A subset of registers are held in a small, fast register file which is usually accessed first. A backing store holds all registers and is only accessed if the desired register is not available in the register cache. These schemes rely on the bypass network to provide most operands to consuming instructions.

This section describes work performed using a register cache or similar structure and works concerned with banking the register file.

3.2.4.1 Cruz *et al.* (2000)

Cruz *et al*. propose different approaches to organising a register file cache of two levels [24]. They explore two caching policies and two mechanisms for fetching values from the lowest cache level to the highest. Results show that the best scheme only caches results that are not read from the bypass network, giving 10% higher IPC compared to a 2 cycle register file with a single level of bypass, but is 8% slower when full bypassing is employed in the comparison register file.

3.2.4.2 Balasubramonian *et al.* (2001)

Balasubramonian *et al*. create a two-level register file, allocating registers from the first level when instructions are renamed and moving them to the second level when all consumers have started execution [13]. The authors also experiment with splitting the registers into separate banks, each with one read and one write port.

Results for the two-level register file show a reduced access time for the first level over a monolithic structure, improving instructions per second by 17%. A banked, single-ported

register file reduces access time by a factor of 2 and energy by a factor of 18. However, combining the two techniques does not bring performance improvements.

3.2.4.3 Butts and Sohi (2004)

Register caching is implemented by Butts and Sohi through decoupled indexing [17]. This means that the cache set index is used to access the cache instead of using the physical register tag. A downside of this is that the cache set index must be held wherever the physical register tag is and the register cache must hold the full physical register tag for comparison on an access. A predictor estimates whether a result will be read from the cache as well as the bypass network and stores it accordingly.

3.2.4.4 Borch *et al.* (2002)

Borch *et al*. introduce microarchitecture loops which occur whenever the result of a computation is needed in an earlier stage of the pipeline [14]. They examine the load resolution loop in detail which exists because a cache hit is always assumed on a load. Then they propose to shorten this loop by having a register cache at each cluster of functional units. Instructions can read their arguments from the register file at dispatch and from the bypass logic or one of the caches at issue. If there is a cache miss then the downside of this scheme is a one cycle penalty to access the register file before the instruction can be executed.

3.2.4.5 Hu and Martonosi (2000)

Hu and Martonosi propose the Value Ageing Buffer to service temporally close register accesses rather than going to the register file [41]. Read operations first go to this buffer but if they are not there then a one cycle penalty is incurred whilst the register file is consulted. Results show a performance loss of 5% and 30% register file power savings for a buffer of 16 entries and a register file of 64 entries.

3.2.4.6 Postiff *et al.* (2001)

Postiff *et al*. present an implementation of a large logical register file [69]. The logical register file is used to store committed values whereas a smaller physical register file is used to store the results of in-flight instructions. The physical register file is directly indexed and keeps its contents even after instructions commit. It is also large enough to contain all results of in-flight instructions to avoid a deadlock situation. This scheme is proposed to allow large logical

	Dynamic Instructions main:	proc:	Killed
a:	= r3, r7		
b:	kill r3		r3 (E–DVI)
c:	call proc		r7 (I–DVI)
d:		r7 =	
e:		= r7	
f:		return	r7 (I–DVI)

Figure 3.2. Explicit and implicit DVI, as defined by Martin *et al.* [58].

register files that the compiler can exploit. However, this would require a fundamental change to the instruction set architecture (ISA) as, as such, is not applicable to current processors.

3.2.4.7 Critical Analysis

This section has presented register caching schemes. These compensate for a large, multi-cycle register file by providing a smaller, fast cache of register values. However, there is usually a cycle penalty to pay if the required value is not found in the cache as reads must then access the actual register file. Deciding the values to cache and how to evict them is important in achieving high performance but can add complexity to the issue and register read pipeline stages.

3.2.5 Compiler Schemes

There have been several attempts to use the compiler to relieve register pressure. This section describes them.

3.2.5.1 Martin *et al.* (1997)

Martin *et al*. implement early register releasing using the compiler to calculate dead value information (DVI) and pass it on to the processor [58]. DVI can be passed via new explicit DVI (E-DVI) instructions which contain a bit-mask indicating the registers released, or implicitly on certain instructions. Procedure calls and returns use implicit DVI (I-DVI) such that when a dynamic call or return is committed the caller-saved registers are released early because the calling conventions implicitly state that they will not be live. The register map table records whether a register is live or has been released early. Figure 3.2 shows an example of both E-DVI and I-DVI instructions.

Results show that DVI can greatly improve performance when a small register file is used. However, these schemes do not have any recovery mechanism in place for handling precise interrupts or exceptions.

3.2.5.2 Lo *et al.* (1999)

Lo *et al*. add compiler and operating system (OS) support to simultaneous multithreaded (SMT) processors to allow registers to be released early [52]. The OS optimisation provides a new, privileged instruction which frees all registers used by a certain context. This is called after a thread terminates so as to free up its registers for the remaining contexts, before another thread starts to execute in the current context.

Five compiler-based schemes are also proposed to release registers early relying on new instructions or special versions of existing ones. Results show a 60% performance improvement for a small register file with four threads running. Again, the schemes presented here do not implement precise interrupts or exceptions.

3.2.5.3 Lozano and Gao (1995)

Lozano and Gao reduce register pressure using the compiler by preventing certain results from being copied from the reorder buffer to the register file on instruction commit [54]. The compiler marks values with short live-ranges that occur within a basic block and assigns a symbolic register as the destination, instead of a normal logical register. Inside the processor the symbolic register is mapped to the reorder buffer position the defining instruction occupies.

Results show that with a reorder buffer of 32 entries, 90% of all writes to the register file are eliminated. However, no performance comparison is made with the baseline for this situation. The authors also do not address problems with implementing precise interrupts and exceptions, nor do they explain how the symbolic registers are encoded in the instruction set architecture (ISA).

3.2.5.4 Critical Analysis

This section has presented compiler-based schemes to optimise the register file. However, all schemes suffer from the fact that the do not properly implement precise interrupts or exceptions.

3.3 Summary

This chapter has described related work in optimising the issue queue and register file. Firstly, schemes which reduce the number of wakeups in the issue queue have been presented, followed by those that gate off pipeline stages when there is little impact on performance. Schemes that calculate the latency of instructions as they wait in the issue queue and those that schedule based on dependence chains have been described, followed by compiler techniques that typically target VLIW architectures.

Register file optimisations have been presented, starting with those that optimise based on the prevalence of narrow-width values within programs. Schemes that allocate registers late, or release early have been described, along with those reducing the port requirements of the register file or those providing a form of register caching. Finally, previous compiler research into register file optimisations has been presented.

Chapter 4

Infrastructure

This chapter describes the benchmarks, compiler and simulation environment used for experimentation in this thesis. It is structured as follows. Section 4.1 describes the selected benchmarks and the way in which they were simulated. Section 4.2 describes the compiler, optimisation passes and the process of compiling a benchmark. Section 4.3 describes the simulator, changes made and the method by which energy results were obtained. Finally, section 4.4 summarises this chapter.

4.1 Benchmarks

The set of benchmarks chosen is the Standard Performance Evaluation Corporation (SPEC) CPU2000 suite [78]. These benchmarks are the standard for research into superscalar microarchitecture optimisations.

The floating point benchmarks from this suite are written in both C and Fortran, some with language extensions. Unfortunately, the SUIF compiler (see section 4.2) had difficulties producing valid code for most of these so instead of using the whole suite only the integer benchmarks were used. However, one of these, *eon*, is written in C++ which could not be compiled either, thus eleven benchmarks were available for simulation.

The benchmarks were run with *ref* inputs for 100 million instructions, skipping the initialisation part and warming the caches and branch predictor for an initial 100 million instructions. This was to make the fast-forwarded simulations more realistic, avoiding incurring a large number of branch mis-predictions and cache misses at the start of the simulations which can occur if the branch predictor and caches are not warmed first. A summary is given in table 4.1.

Benchmark	*Initial fast-forward*	*Description*
164.gzip	30000000	Compression
175.vpr	260000000	FPGA Circuit Placement and Routing
176.gcc	10000000	C Programming Language Compiler
181.mcf	1270000000	Combinatorial Optimisation
186.crafty	20000000	Game Playing: Chess
197.parser	520000000	Word Processing
253.perlbmk	10000000	PERL Programming Language
254.gap	130000000	Group Theory, Interpreter
255.vortex	1500000000	Object-oriented Database
256.bzip2	30000000	Compression
300.twolf	490000000	Place and Route Simulator

Table 4.1. Summary of benchmarks.

4.2 Compiler

The compiler analysis was written as a pass in Machine SUIF [77] from Harvard, an extension to the SUIF compiler from Stanford [79]. SUIF provides the compiler front-end and passes to translate C programs into an intermediate representation (IR) whilst Machine SUIF provides back-end optimisation, code generation and register allocation passes. Each pass works on a per-procedure basis with libraries provided to automatically translate the IR into a control-flow graph enabling data-flow and liveness analysis.

The whole compiler is highly modular with few constraints on the ordering of passes. All passes in this thesis were applied at the final stage, just before writing code to a file. In effect, they operated on assembly code in order to know exactly which instructions were going to be executed by the processor.

After compilation, the assembly code files were assembled and linked with the native Alpha tools through the Alpha compiler with no further optimisation applied.

An overview of the whole compilation process is shown in figure 4.1. Code generation occurs after translation to the intermediate representation. Following this is the expansion of pseudo-instructions, which are instructions provided by the assembler as shorthand for commonly-used groups of other instructions. For example, a byte load instruction can be expanded into five or six separate instructions that load a 64-bit value and extract the correct byte using masks and shifts. Normally the assembler would expand these pseudo-instructions. However, as the analyses proposed in this thesis require exact knowledge of the control flow and data flow graphs, a compiler pass performs the expansions instead.

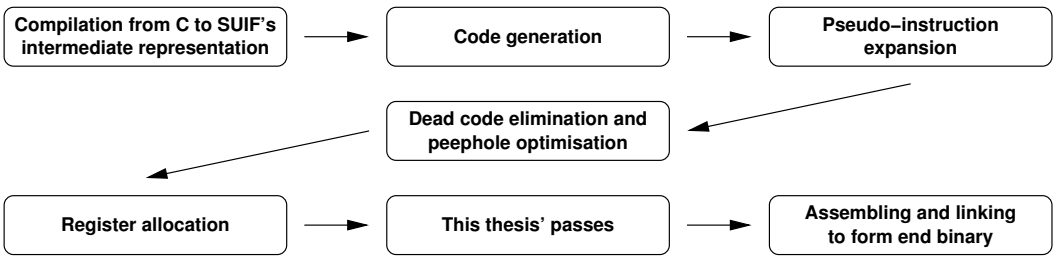


Figure 4.1. Overview of compilation.

The dead code elimination pass removes all instructions that produce unneeded values, along with unnecessary branches. Peephole optimisation removes unneeded register-move instructions and propagates constants forward whenever possible. Following these optimisations, register allocation is performed.

The passes proposed in this thesis occur immediately before assembling and linking of the code to produce the final binary. They are performed at this stage so that the instructions that are analysed are exactly the same as those that are executed by the processor.

4.3 Simulation Environment

This section presents the simulator used in all experiments. Firstly the baseline simulator is described and the changes made to the microarchitecture are discussed. Following that the energy model is described, from which the energy results were obtained.

4.3.1 Simulator

The benchmark programs were run using Wattch [15], based on the SimpleScalar simulator [16]. This was chosen because it can model a high-performance superscalar processor and provides power results. It is also the standard environment for research into superscalar microarchitecture.

Various changes had to be made to SimpleScalar in order to carry out research into the issue queue and register files. SimpleScalar uses a register update unit to hold instructions and data as it passes through its five-stage pipeline. This was altered to consist of a reorder buffer, issue queue and separate integer and floating point register files. The issue queue and register files were split into banks of 8 entries, each of which could be turned off independently of the

Parameter	*Configuration*
Fetch, decode, issue and commit width	8 instructions
Branch predictor	Hybrid 2K gshare, 2K bimodal, 1K selector
BTB	2048 entries, 4-way
L1 Icache	64KB, 2-way, 32B line, 1 cycle hit
L1 Dcache	64KB, 4-way, 32B line, 2 cycles hit
Unified L2 cache	512KB, 8-way, 64B line, 10 cycles hit, 50 cycles miss
Reorder buffer	128 entries
Issue queue	80 entries
Integer register file	112 entries (14 banks of 8)
FP register file	112 entries (14 banks of 8)
Integer FUs	6 ALU (1 cycle), 3 Mul (3 cycles)
FP FUs	4 ALU (2 cycles), 2 MultDiv (4 cycles mult, 12 cycles div)

Table 4.2. Processor configuration.

others to save both dynamic and static energy, as discussed in section 4.3.2.

The pipeline depth was increased by seven extra fetch stages to better model the long pipelines of modern superscalar processors. Finally, the branch predictor was altered such that it was queried at fetch and updated at writeback. Mis-speculated unconditional branches are resolved in a late decode stage whereas other mis-predictions (conditional branches and jump instructions with an incorrect target) are squashed at writeback, rather than in the original version where some mis-predicted instructions were squashed at dispatch. This was to make the branch predictor operate in a more realistic manner.

The configuration of the processor was designed to be typical of a high-performance superscalar supporting 8-wide fetch, decode, issue and commit, the details of which can be seen in table 4.2. The register files are slightly smaller than the reorder buffer because some instructions do not have a destination register. When the reorder buffer is full, it is likely that it contains some of these instructions, hence fewer physical registers are required as rename buffers than there are entries in the reorder buffer. Although this means that occasionally the register file must stall dispatch due to a lack of free registers, there are fewer redundant registers when the reorder buffer is the bottleneck to dispatch. The issue queue size is consistent with processors allowing a similar number of in-flight instructions [80].

4.3.2 Energy Measurements

This section discusses the energy model used with the simulator presented in section 4.3.1.

Operation	*Components*	*Energy (pJ)*
Read	(rf_addr+4to16+rf_out) +(3to8+rf_wl+rf_bl+rf_data)∗banks_on	32.7 +17.4∗banks_on
Write	(rf_addr+4to16+rf_wl+rf_bl) +(3to8+rf_data)∗banks_on	10.2 +12.3∗banks_on

Table 4.3. Energy consumed in the register file for each type of access [4, 2]. For a breakdown see appendix A.

Estimating power dissipation, or energy consumption, within a superscalar processor is a difficult task. Several models and tools to accomplish this do exist, but their results can vary dramatically, as experienced by Ghiasi and Grunwald [33]. They find that results from the power models are often statistically indistinguishable from the baseline, even with a 90% confidence level. The authors also find that results from two separate tools can lead to different conclusions being drawn from the same set of experiments. Following on from this, Kim *et al.* describe alterations that can be made to a cycle simulator to produce meaningful power figures [43]. However, current superscalar simulators have yet to incorporate these changes.

The dynamic and static energy models used in this thesis were developed by Abella and González [5, 4, 2] from those used by Wattch [15] and Cacti [73], where Wattch is an extension to SimpleScalar [16] and Cacti is a tool for calculating the area and power of memory circuits. Although primarily designed for cache circuits, Cacti can be used to model the banked memories in this issue queue and register file since their layouts are similar. There is also no alternative tool targeted towards these processor components that could be used.

The dynamic energy consumption of each register file operation is shown in table 4.3 and likewise with the issue queue in table 4.4, taken from the model by Abella and González [5, 4, 2]. The breakdown of energy for each component is shown in appendix A.

Due to the banked nature of the issue queue and register file, accessing an entry is parallelised which provides a fast access time but high energy consumption. On a read, the entry decoding (3to8), reading (rf_wl, rf_bl) and driving data out of each bank (rf_data) are performed in each bank that is turned on. The correct register is selected from these (rf_addr, 4to16) and driven out of the register file (rf_out). When writing, data is sent to all banks and an entry decoded in each, but the actual writing is not parallelised to avoid data corruption.

Within the issue queue, to wake operands without the matchline gating proposed by Folegnani and González [32], all matchlines (ml) and operand bitlines (opnd_bl) are driven in each bank that is turned on and the operand wordline (opnd_wl) only driven on a successful match. The separate data types (opcode, operands and tags) are accessed independently with the ad-

Operation	*Components*	*Energy (pJ)*
Selection	None	123.6
Wakeup	(tagl+8∗ml+opnd_bl) ∗ 2∗banks_on +opnd_wl∗matches+gtagl	19.6∗banks_on +0.5∗matches+2.1
Source tag write	(iq_addr+4to16+stag_wl+stag_bl) +(3to8+stag_data)∗banks_on	5.2 +2.4∗banks_on
Opcode read	(iq_addr+4to16+opc_out) +(3to8+opc_wl+opc_bl+opc_data)∗banks_on	10.9 +6.8∗banks_on
Opcode write	(iq_addr+4to16+opc_wl+opc_bl) +(3to8+opc_data)∗banks_on	6.4 +5.1∗banks_on
Operand read	(iq_addr+4to16+opnd_out) +(3to8+opnd_wl+opnd_bl+opnd_data)∗banks_on	32.3 +18.0∗banks_on
Operand write	(iq_addr+4to16+opnd_wl+opnd_bl) +(3to8+opnd_data)∗banks_on	10.0 +12.7∗banks_on
Destination tag read	(iq_addr+4to16+dtag_out) +(3to8+dtag_wl+dtag_bl+dtag_data)∗banks_on	6.3 +2.9∗banks_on
Destination tag write	(iq_addr+4to16+dtag_wl+dtag_bl) +(3to8+dtag_data)∗banks_on	5.2 +2.4∗banks_on

Table 4.4. Energy consumed in the issue queue for each type of access [5, 2]. For a breakdown see appendix A.

dress routing and entry decoding logic duplicated to avoid including complex hardware to select only parts of a line.

The model used in this thesis for static energy consumption is based on one developed by Abella and González [5, 4, 2] which takes into account the size and ports of each of the components in the issue queue and register file. This was chosen because it is closely related to the dynamic energy model and allows direct comparisons with work performed by Abella and González, whose issue queue limiting scheme is compared with in chapter 5. Furthermore, both the static and dynamic energy consumption is affected by the number of issue queue or register file banks turned on, which is determined by the occupancy of each structure. A low occupancy increases the opportunities for turning banks off, hence occupancy reductions for each structure are also presented with the energy savings achieved. Therefore, it is expected that usage of a different model, such as HotLeakage [87], would show similar energy savings to those presented.

4.4 Summary

This chapter has explained the infrastructure used in this thesis. The SPEC CPU2000 benchmark suite [78] was chosen and compiled using the Machine SUIF compiler [77, 79] to generate Alpha assembly code files from which binaries were created using the native Alpha assembler and linker. These were then simulated using an adapted version of Wattch [15] and SimpleScalar [16].

Chapter 5

Issue Queue Energy Savings

Superscalar processors contain complex control logic in order to aggressively issue instructions through extraction of sufficient instruction level parallelism (ILP). Unfortunately, those structures used to perform out-of-order execution consume a large amount of energy which has important implications for future processor technology.

The issue logic is one of the main sources of energy dissipation in current superscalar processors [25]. It has been estimated that up to 27% of the energy consumed by a processor is in the issue logic [32]. Furthermore, the issue logic is one of the components with the highest power density and is a hot-spot. Reducing its energy consumption is, therefore, more important than for other structures.

There has been much work in developing hardware techniques to reduce the energy cost of the issue logic by turning off unused entries and adapting the issue queue size to the available ILP [32, 21, 5]. Unfortunately, there is inevitably a delay in sensing rapid phase changes and adjusting accordingly. This leads to either a loss of performance due to too small an issue queue or excessive power dissipation resulting from too large a queue.

This chapter proposes an entirely different approach: software-directed issue queue control. In essence, the compiler knows which parts of the program are to be executed in the near future and can resize the queue accordingly. It reduces the number of instructions in the queue without delaying the critical path of the program. Reducing the number of instructions in the issue queue reduces the number of non-ready operands woken each cycle and hence saves energy [32].

Correctly sizing the issue queue reduces the number of non-critical in-flight instructions and so reduces the number of entries needed in the register file. This enables further energy

savings by simply turning off empty banks. The energy savings using the schemes described in this chapter are presented in sections 5.4 and 5.5.

The rest of this chapter is structured as follows. Section 5.1 presents an example showing how energy savings can be achieved in the issue queue. Section 5.2 describes the microarchitecture used and the small changes needed by this scheme. This is followed by section 5.3 which describes the model used within the compiler to determine the critical path through a program region and the specialisation of the analysis for directed acyclic graphs and loops. Section 5.4 describes initial, coarse-grained issue queue throttling schemes that determine the queue requirements on a procedure by procedure basis. Refined, fine-grained throttling is presented in chapter 5.5 and is followed by section 5.6 which concludes this chapter.

5.1 Motivation

This section describes the minimising of the issue queue size without affecting the critical path. The critical path of a program depends on its data dependence structure but is also affected by the limited number of functional units and variable memory latency due to cache misses, amongst other things. For the sake of this example, only data dependences that affect the critical path are considered.

Figure 5.1 shows a basic block where all instructions belong to at least one critical path. To aid readability, instructions are written using pseudo-code. A fragment of assembly code is shown in figure 5.1(a) where instruction a defines register r1 which is then used by instruction b whilst defining r2. This register is then used by instructions c and d who define registers r3 and r4 respectively. Finally, instruction e uses registers r2 and r3 and instruction f uses registers r1 and r4. The data dependence graph (DDG) is shown in figure 5.1(b).

There is no need for instructions b, c, d, e and f to be in the issue queue at the same time as a as they are dependent on it and so cannot issue at the same time as it. Likewise, instructions c, d, e and f do not need to be in the issue queue at the same time as b. In fact, instructions e and f do not need to enter the issue queue until c and d leave. Limiting the issue queue to only 2 instructions means the code will execute in the same number of cycles, but fewer wakeups will occur and so energy will be saved.

Figures 5.1(c) and 5.1(d) show the issue queue in the baseline and limited cases respectively. A dispatch width of 8 instructions is assumed with each instruction taking one cycle to execute. It is also assumed instruction a has no other input dependences and can therefore issue the cycle after it dispatches. Finally, as in Folegnani and González [32], it is assumed

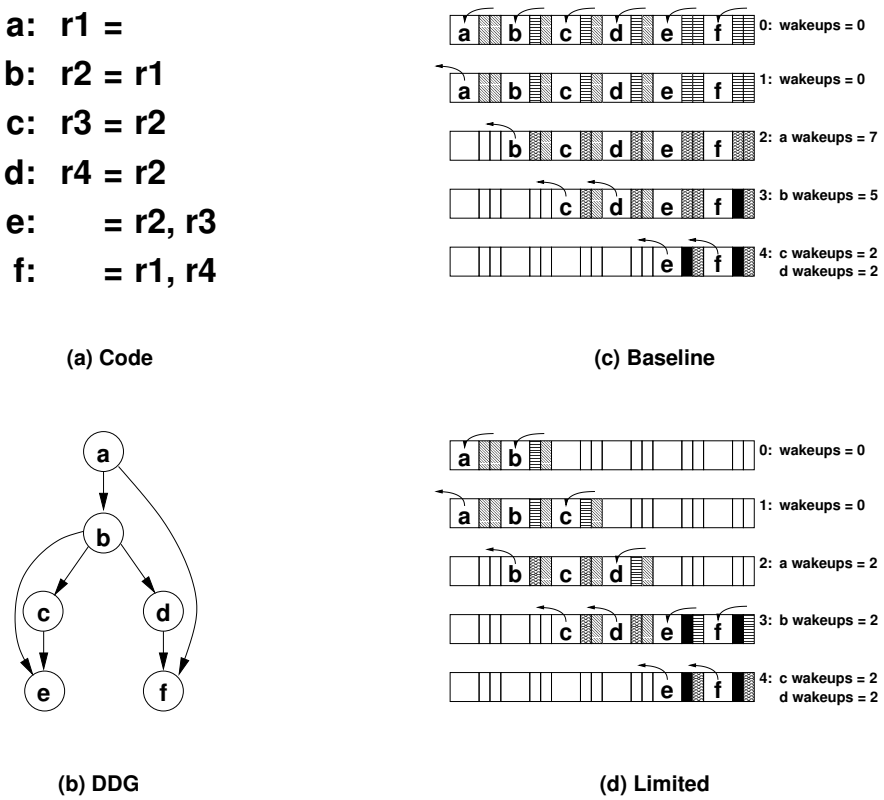


Figure 5.1. Issue queue energy savings. 5.1(a) shows a basic block and 5.1(b) shows its DDG. In 5.1(c) it takes 5 cycles and causes 16 wakeups. Limiting the queue to 2 entries in 5.1(d) means it still takes 5 cycles but only causes 8 wakeups.

that empty and ready operands do not get woken. Arrows denote whether an instruction is dispatched into the issue queue or issued from it. A white rectangle next to an instruction indicates an empty operand with an empty entry while a rectangle with diagonal lines denotes an operand that is not needed. A rectangle with horizontal lines shows an operand yet to arrive and one that is crossed diagonally shows a wakeup on that operand. Finally, a black rectangle indicates an operand already obtained.

In the baseline case, figure 5.1(c), all six instructions dispatch in cycle 0. Instruction a issues in cycle 1, completing execution in cycle 2. It causes seven wakeups and allows instruction b to issue. In cycle 3, b finishes causing five wakeups, allowing instructions c and d to issue. In cycle 4, c and d cause two wakeups each and finally e and f can issue. They write back in cycle 5 and there are sixteen wakeups in total.

Now consider figure 5.1(d) with the same initial assumptions, but with the constraint that

only two instructions can be in the issue queue at any one time. In cycle 0, only a and b are dispatched. Instruction a issues in cycle 1 and this makes way for c to dispatch. In cycle 2, instruction a causes just two wakeups and b issues, allowing instruction d to dispatch. The following cycle it causes two wakeups and allows c and d to issue making way for instructions e and f to dispatch. In cycle 4, c and d cause two wakeups each and the final two instructions issue, writing back in cycle 5. There is no slowdown and only eight wakeups occur, a saving of 50%. In practice the dependence graphs are more complex and resource constraints must be considered, yet this example illustrates the basic principle of the technique.

5.2 Microarchitecture

This section describes the hardware changes needed to support the limiting of the issue queue. The compiler has to pass information to the processor about the number of issue queue entries needed. Throughout this chapter, two methods of accomplishing this are evaluated: passing through special NOOPs and the tagging of instructions.

The special NOOPs consists of an opcode and some unused bits, in which the issue queue size is encoded. The special NOOPs do nothing to the semantics of the program and are not executed, but are stripped out of the instruction stream in the final decode stage before dispatch.

The second method of passing information from the compiler to the processor assumes there are a number of unused bits within each instruction which can be used to encode the issue queue size needed. These instructions are executed as normal, but the issue queue size information they contain is extracted during decode and used when the instruction dispatches. The advantage of using tagging is that there are none of the side-effects associated with special NOOPs, such as an increase in instruction cache misses.

This chapter assumes dispatch-bound operand fetching because much of the literature concerning issue queue optimisations also uses this scheme. In particular, the hardware approach by Abella and González [5] which is compared with in section 5.5.3 uses this type of operand fetching, therefore this chapter also employs it to enable meaningful comparisons.

5.2.1 Issue queue

A multiple-banked issue queue is assumed where instructions are placed in sequential order. We assume that the queue is non-collapsible as in [32, 21, 5]. Having a compaction scheme would cause a significant amount of extra energy to be used each cycle. The queue is similar to

[21] where a simple scheme is used to turn off the CAM and RAM arrays at a bank granularity at the same time. The selection logic is always on but it consumes much lower energy than the wakeup logic [67].

Empty and ready entries within the queue are prevented from being woken by gating off the precharge signal to the CAM cells, as in [32]. An overview of the issue queue modifications is shown in chapter 2 with a summary of the banking and gating of entries. As explained in that chapter, the baseline simulator performs no gating (empty and ready operands are woken) and all issue queue banks are permanently on.

In addition to these changes, the schemes in this chapter limit the number of entries allowed to contain instructions through the use of a special NOOP or by tagging instructions. The changes required for each are explained with the approaches in sections 5.4 and 5.5.

5.2.2 Fetch queue

Once instructions are fetched they are placed in the fetch queue where they spend several cycles being decoded before being dispatched to the issue queue. Each cycle the dispatch logic selects a number of instructions to move from the head of the fetch queue to the issue queue. The selection of instructions to dispatch has to take into account the dispatch width of the processor, availability of issue queue entries and number of free registers, amongst other criteria. Limiting the number of issue queue entries has no effect on the selection logic which simply chooses instructions with all operands ready, as usual.

The initial implementations use special NOOPs to pass limiting information from the compiler to the processor. When a special NOOP is encountered in the instructions to be dispatched, it is removed from the instruction stream and its value used as the new limit. Tagged instructions are not removed from the stream but the limiting value they contain is decoded and used as before.

5.2.3 Register Files

Every newly dispatching instruction with a logical destination register is allocated a physical register from one of the register files. There is one integer and one floating point register file in the processor used. Limiting the number of instructions entering the issue queue reduces the number of instructions dispatching along mis-speculated paths and hence reduces register pressure. To indirectly save energy in the register files they are banked, enabling them to be

turned off for static and dynamic energy savings. More information on the register file banking scheme can be found in chapter 2.

5.3 Compiler Analysis

This section describes the complier analysis performed to determine the number of issue queue entries needed by each program region. It is used in sections 5.4 and 5.5 for coarse-grained and fine-grained issue queue throttling and is based on simple methods to find the critical path of a program taking into consideration data dependences and resources.

Firstly, the program is split into directed acyclic graphs (DAGs) and loops, as described in section 5.3.1. A dependence graph is built for each structure and section 5.3.2 describes the critical path model that is derived from each one. Sections 5.3.3 and 5.3.4 then describe how the analysis is specialised to determine the minimum number of issue queue entries required by each DAG and loop to avoid slowing down the critical path.

5.3.1 Program Representation

The compiler considers each procedure individually, first building the control flow graph (CFG) using basic blocks as nodes and the flow of control between them as the edges. Analysis is specialised for loops because instructions from different loop iterations are executed in parallel when the loop is run. Hence, new CFGs are created for the loops as explained below, and their backward edges removed in the original. This is to preserve the dependences between instructions within each loop and those following. No inter-procedural analysis is performed, so the two program structures created are DAGs and loops.

The algorithm for creating a DAG starts with the first node in the procedure's CFG (with each loop's backward branches removed) and continually adds nodes to the DAG until a call is encountered. Because no inter-procedural analysis is performed, a new DAG is then created using the first node after the call as the entry point. As an optimisation, DAGs with common nodes can be merged together to create a single DAG with multiple starting nodes.

Loops are created from the natural loops in the procedure, which are loops with a single entry point. These are found using a method based on algorithm 10.1 in from Aho *et al*. [8]. For each level of loop nesting a new control flow graph is created and each is considered by itself, starting from the innermost loop. Nodes in previously seen loops are removed from the new CFG so that they are not analysed more than once. Due to the lack of inter-procedural

1. Create the procedure's control flow graph, CFG
2. For each node, N, in CFG after a call
   (a) Create a new directed acyclic graph, DAG, containing N
   (b) While there is a node in DAG, M, which does not end in a call
      i. For each successor node, S, of M
         (1) Add S to DAG
3. Find natural loops
4. For each loop starting with the innermost
   (a) Build its control flow graph, CFG
   (b) Remove nodes in previously seen loops
   (c) Remove nodes ending in a call
   (d) Remove nodes which are not part of a cycle in CFG

Figure 5.2. Algorithm for forming DAGs and loops from a procedure. A new DAG is formed after every procedure call. Each loop has a new control flow graph created using the loop nodes. Nodes that should not be part of the loop are then removed.

analysis, nodes ending in calls are also removed. Finally, it must be possible to traverse the loop's CFG from any node and eventually end up back at that node, therefore nodes that do not satisfy this requirement are also removed.

The algorithm for creating DAGs and loops is shown in figure 5.2 and an example procedure shown in figure 5.3. The backward branch from the loop is removed and the DAG formed from all basic blocks, because there are no procedure calls. A new control flow graph is created for the loop.

Once the procedure has been split into DAGs and loops, the dependence graphs, from which critical path information can be determined, can be created. Section 5.3.2 describes this process, then sections 5.3.3 and 5.3.4 describe how the analysis is specialised for DAGs and loops respectively.

5.3.2 Critical Path Model

The critical path is modelled as a dependence graph, similar to those proposed by Tullsen and Calder [84] and Fields *et al*. [30]. This is because it provides an accurate representation of the events that occur to instructions as they are dispatched and issued. The model differs from Fields *et al*. [30] in two ways. Firstly it is statically created by the compiler instead of being dynamically determined in the processor and, secondly, it does not have nodes for the commit of instructions. It does, however, have extra nodes for load and store instructions which correspond to events that occur in the load/store queue. The commit nodes are removed

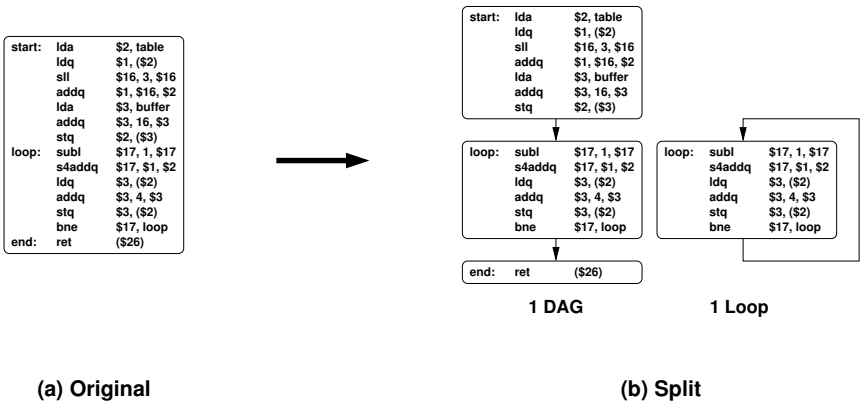


Figure 5.3. An example procedure and the result of splitting into a DAG and a loop. There are no procedure calls in the original code so only one DAG is created with the loop's backward edge removed. There is also a new control flow graph created for the single loop.

because the model only needs to determine the critical path as instructions pass through the issue queue, whereas Fields *et al*. [30] determine the critical path through the whole pipeline.

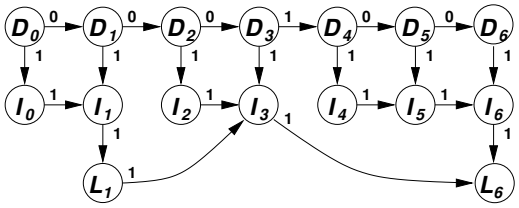
In the model used in this thesis, each instruction, i, is represented by at least two nodes (loads and stores are represented by three) which correspond to events occurring to the instruction. There is a D (dispatch) node for when the instruction is dispatched into the issue queue, an I (issue) node for when the instruction is issued from the queue and an L (load/store) node if the instruction is a load or store, which denotes the issue of the instruction from the load/store queue.

The edges connecting the graph denote dependences between nodes. Each edge is weighted with the minimum number of cycles that the dependence takes to be resolved. Unlike Fields *et al*., control dependences between branches are not modelled because it is assumed that all branches will be predicted correctly. Figure 5.4 shows an example piece of assembly code and the dependence graph that is formed as the critical path model.

The dependences modelled are shown in table 5.1. The first is an edge between consecutive D nodes to model in-order dispatch of instructions. The second is an edge between an instruction's D and I nodes which always has a weight of 1 because all instructions must enter the issue queue and the following cycle is the first in which they can be considered for issuing. Similarly, for loads and stores, there is an edge with weight 1 between the I and L nodes. In this case the I node performs the address calculation, taking 1 cycle, then the L node represents

```
start:  lda     $2, table
        ldq     $1, ($2)
        sll     $16, 3, $16
        addq    $1, $16, $2
        lda     $3, buffer
        addq    $3, 16, $3
        stq     $2, ($3)
```

(a) Code



(b) Dependence graph

Figure 5.4. An example piece of assembly code and its dependence graph. Edges are weighted with the latency (in cycles) taken to resolve the dependence. A *D* node with latency 0 on its incoming edge can dispatch at the same time as its predecessor, whereas a *D* node with latency 1 must be dispatched a cycle after its predecessor. In this example, all operations are assumed to take just one cycle.

the actual load or store.

In the simulated processor, loads are only allowed to issue from the load/store queue before previous stores once the addresses of both have been calculated. To model this, edge 4 is used to create a dependence between the *I* node, which is the address calculation, of a store and the *L* nodes of each subsequent load. When the address of a load is the same as that of a previous store, the *L* nodes are connected with an edge of weight 1, which is the number of cycles required for store forwarding in the processor, as shown by edge 7.

Data dependence edges are shown in the table too. A load's result is computed by the *L* node whereas in all other instructions it is the *I* node that defines the output register. Stores are a special case because their address source register is used by the *I* node whereas the data source register is used by the *L* node. Hence there are extra edges 6 and 9 in the table.

When adding edges to the dependence graph, conservative assumptions are made except in the following cases: all branches are assumed to be predicted correctly; all loads are assumed to hit in the first level data cache; and where a load follows a store and it cannot be determined that they access the same memory address, it is assumed that they do not and that the load can issue before the store once both instructions' addresses have been calculated.

The whole process of creating the dependence graph is shown as an algorithm in figure 5.5. The *D* and *I* nodes are created first and edges placed between them. Loads and stores have to have *L* nodes created too.

Id	*Constraint*	*Edge*	*Notes*
1	In-order dispatch	$D_p \rightarrow D_i$	If p is an immediate predecessor of i
2	IQ issue follows dispatch	$D_i \rightarrow I_i$	For every instruction
3	LSQ issue follows IQ issue	$I_i \rightarrow L_i$	For every load and store
4	No speculative load bypassing	$I_p \rightarrow L_i$	If i is a load and p is a previous store
5	Data dependence	$I_p \rightarrow I_i$	If previous non-load p defines a source register of i
6	Data dependence	$I_p \rightarrow L_i$	If previous non-load p defines the data register of store i
7	Store forwarding	$L_p \rightarrow L_i$	If previous store p has the same address as load i
8	Data dependence	$L_p \rightarrow I_i$	If previous load p defines a source register of i
9	Data dependence	$L_p \rightarrow L_i$	If previous load p defines the data register of store i

Table 5.1. Edges present in the critical path model.

A dependence graph is created for each DAG and loop within the procedure. This graph can be used to calculate the issue queue requirements of the program structure being analysed. The way in which it is used is specialised according to this. Section 5.3.3 describes the analysis for DAGs and then section 5.3.4 details it for loops.

5.3.3 Specialised DAG Analysis

Once the dependence graph has been formed each DAG is considered separately. Section 5.3.3.1 describes the specialised analysis to find the issue queue requirements of each DAG and section 5.3.3.2 provides a detailed example of this.

5.3.3.1 Analysis

Starting with the entry point, the DAG's dependence graph is iterated over to determine the number of issue queue entries needed. In this stage of the algorithm two further constraints are modelled that could not be included in the initial representation of the graph, namely functional unit contention and a limited issue width of the processor.

As the graph is iterated over, the set of nodes reached is recorded, along with the iteration

1. For each instruction i
   (a) Create nodes D_i and I_i and insert edge $D_i \rightarrow I_i$
   (b) For each immediate predecessor instruction p of i
      i. Insert edge $D_p \rightarrow D_i$
   (c) If i is a load or store
      i. Create node L_i and insert edge $I_i \rightarrow L_i$
   (d) For each previous non-load p defining a source register r of i
      i. If i is not a store or r is not i's data register
         (1) Insert edge $I_p \rightarrow I_i$
      ii. Else
         (1) Insert edge $I_p \rightarrow L_i$
   (e) For each previous load p defining a source register r of i
      i. If i is not a store or r is not i's data register
         (1) Insert edge $L_p \rightarrow I_i$
      ii. Else
         (1) Insert edge $L_p \rightarrow L_i$
2. For each load instruction i
   (a) For each predecessor store instruction p of i
      i. Insert edge $I_p \rightarrow L_i$
      ii. If i accesses the same address as p
         (1) Insert edge $L_p \rightarrow L_i$

Figure 5.5. Algorithm for building the dependence graph for a DAG or loop.

number they were reached on. This is called the issue set. Also recorded is the oldest I node which has not been added to the issue set at the start of each iteration.

To model a finite issue width, only a certain number of I and L nodes can be added to the issue set on any iteration, which corresponds to the issue width of the processor being compiled for (in the processor used, 8 instructions can be issued each cycle). A new edge is inserted into the dependence graph from the last node that was added to the issue set to each node that cannot be added due to the finite issue width, giving it a weight of 1. This has the effect of preventing the I or L node from being considered for inclusion until the next iteration.

Functional unit contention is modelled in a similar manner using knowledge of the number of each type of functional unit available, along with the type needed by each node. The number of nodes that use each functional unit type are counted each iteration to a maximum of the number available, then further nodes needing the same type are prevented from being placed in the issue set. Those that should be added but are prevented due to the quota for the required functional unit having been achieved have a new edge added as before.

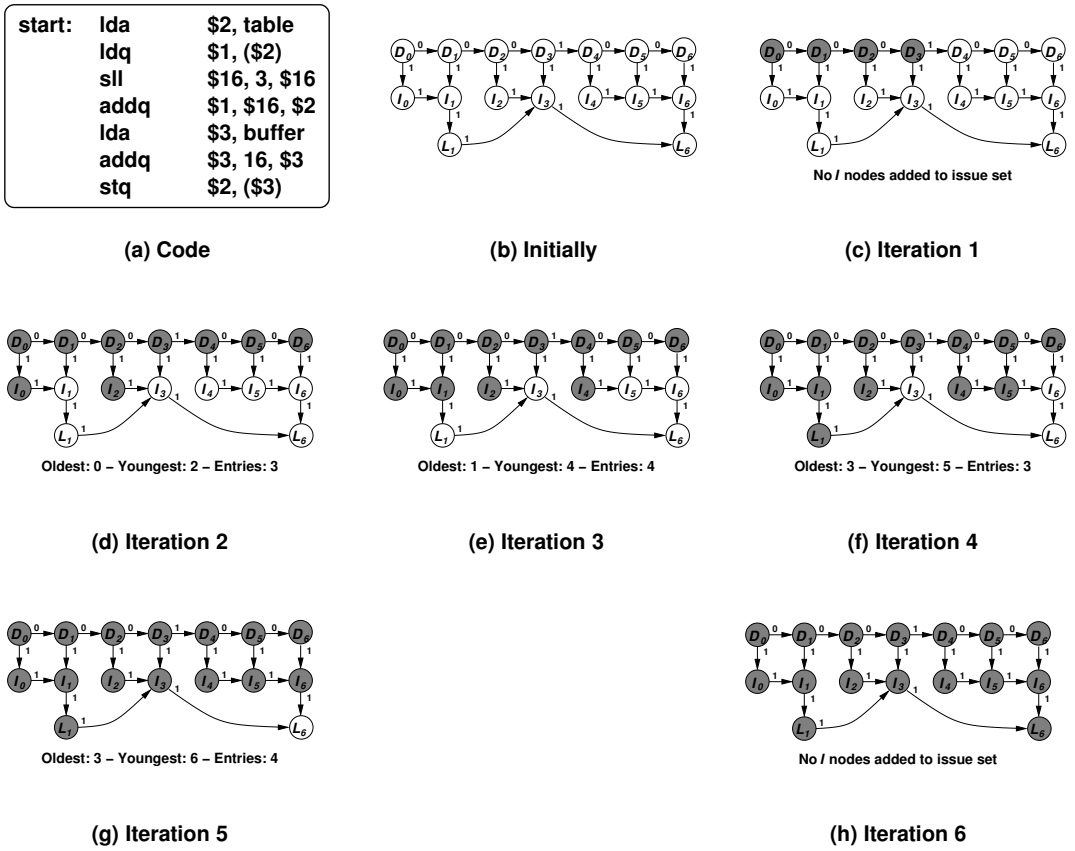
1. $next\_nodes = \{(D_0, 0)\}$
2. While $next\_nodes \neq \emptyset$
   (a) $issued = 0$
   (b) $oldest\_inode =$ oldest I node not in $issue\_set$
   (c) For each functional unit type T
      i. $used(T) = 0$
   (d) For each pair $(N, X) \in next\_nodes$
      i. If $X = 0$
         (1) If $issued < issue\_width$ and $used(FU(N)) < number(FU(N))$
            a. Then $issue\_set = issue\_set \cup N$
            b. $used(FU(N)) = used(FU(N)) + 1$
            c. $issued = issued + 1$
            d. $youngest\_inode = Younger(N, youngest\_inode)$
            e. For each edge with weight W connecting N with successor S
               (i) $next\_nodes = next\_nodes \cup (S, W)$
      ii. Else
         (1) $X = X + 1$
   (e) $entries = MAX(entries, Distance(oldest\_inode, youngest\_inode))$

where $FU(N)$ is the functional unit type required by node N
$Younger(M, N)$ returns the younger of nodes M, N
$Distance(M, N)$ returns the number of entries between nodes M, N

Figure 5.6. Algorithm for analysing a DAG.

On each iteration the dependence graph is traversed along edges from nodes that have already been included in the issue set to those nodes that are not yet included. The edge weights determine the number of iterations to wait after a node has been added to the set before the edge can be traversed and the dependence satisfied.

The whole graph is iterated over until all nodes are included in the issue set. The oldest I node not in the issue set at the start of each iteration is recorded, along with the youngest that is added to the set over the course of the iteration. The difference between the two gives the issue queue size needed on that iteration to prevent a slowdown of the critical path. The maximum size over the whole iteration gives the maximum issue queue size needed for the whole dependence graph. Figure 5.6 describes the complete algorithm for DAG analysis.



(a) Code (b) Initially (c) Iteration 1

(d) Iteration 2 (e) Iteration 3 (f) Iteration 4

(g) Iteration 5 (h) Iteration 6

Figure 5.7. Example of DAG analysis with a dispatch and issue width of 4 and no limit on the number of functional units. With the issue queue limited to four entries, this DAG would not be slowed down at all.

5.3.3.2 Example

Figure 5.7 shows an example of a piece of code, its dependence graph and the analysis applied to it. The initial graph is shown in figure 5.7(b). On the first iteration, shown in figure 5.7(c) the D nodes are added to the issue set. To indicate this they are coloured grey. However, because the dispatch width allows a maximum of four instructions to enter the pipeline, node D_4 is prevented from being included so the edge $D_3 \rightarrow D_4$ has a weight of 1.

The second iteration is shown in figure 5.7(d). The final three D nodes are added to the issue set along with nodes I_0 and I_2 which have no input dependences. At the start of the iteration I_0 is the oldest I node not in the issue set and the youngest I node to be included in the set is I_2 with a distance of 3 between them.

On the third iteration (figure 5.7(e)) the oldest I node beforehand is I_1 and the youngest to be included in the issue set is I_4 with a distance of 4 between them. On the fourth iteration (figure 5.7(f)), three entries are needed again, although I_3 is not added to the issue set. It is added on iteration five (figure 5.7(g)) when four entries are needed, then finally, as shown in figure 5.7(h), the last node is added to the issue set and the analysis is finished. The maximum number of entries needed over all iterations in this example is just four. This would allow the DAG to issue without slowing down the critical path, but would prevent some instructions being dispatched to the queue before they can be executed.

5.3.4 Specialised Loop Analysis

Out-of order execution of loops allows instructions from different loop iterations to be executed in parallel leading to a pipeline parallel execution of the loop as a whole. The analysis, therefore, has to be adjusted accordingly. Section 5.3.4.1 describes this whilst section 5.3.4.2 provides a detailed example.

5.3.4.1 Analysis

Cycles containing I and L nodes in the dependence graph are detected and that with the longest latency chosen. This set of nodes is called the cyclic dependence set of nodes (CDS) and is this set of instructions that is the critical path. This is because there is usually less instruction-level parallelism within a loop than the number of instructions that can be dispatched each cycle. Hence fewer instructions are issued than dispatched, so the issue queue fills with instructions from many loop iterations. Others cannot be dispatched until older instructions (those from previous iterations) issue. The CDS dictates the length of time each loop iteration takes to be completely issued and the next started, hence this affects the dispatch of new instructions into the queue. Slowing it down prevents new instructions being dispatched, hence this is the critical path.

Equations are formed for each I and L node based on the relationships within the dependence graph. Within each equation there is the name of a dependent node and the loop iteration it belongs to. The equations express the minimum number of cycles after the dependent node has issued before the current node can issue when the loop is actually executed. By substitution, these equations can be manipulated to express each I and L node in terms of a node within the CDS, meaning that relationships between CDS nodes and others within the graph are exposed. From these new equations it is possible to determine the nodes (possibly on different

1. $CDS = Find\_CDS(nodes)$
2. For each node N in $nodes$
   (a) For each immediate predecessor node P in $nodes$
      i. Form equation $N_0 = P_i + Latency(P_i)$
3. While there's a change in the equations and an equation not related to a CDS node
   (a) For each node N in $nodes$
      i. For each equation E of the form $N_i = R_j + X$
         (1) If $\exists$ equation $R_k = C_l + Y$ where $C \in CDS$
            a. Rewrite E as $N_i = C_{l+j-k} + X + Y$
4. For each node N in CDS
   (a) For each equation of the form $L_i = N_j$
      i. $oldest\_inode = Older(L_i, N_j, oldest\_inode)$
      ii. $youngest\_inode = Younger(L_i, N_j, youngest\_inode)$
   (b) $entries = MAX(entries, Distance(oldest\_inode, youngest\_inode))$

where $Find\_CDS(graph)$ returns the cycle with the highest weight in $graph$
$Latency(N)$ returns the latency of N
$Older(M,N,O)$ returns the elder of nodes M,N,O
$Younger(M,N,O)$ returns the younger of nodes M,N,O
$Distance(M,N)$ returns the number of entries between nodes M,N

Figure 5.8. Algorithm for analysing a loop.

loop iterations) that could issue together when the loop is executed within the processor. The issue queue size needed is calculated from the largest distance between any two I nodes that can issue together.

Figure 5.8 summarises the algorithm for analysing a loop. The algorithm is guaranteed to terminate in step 3 because the termination conditions are either that all nodes are related to a CDS node, or that there is no change in the equations. Whenever an equation is altered, progress is made towards the first termination condition. Hence, either no progress is made and the algorithm terminates due to the second condition, or progress is made and the first condition is met, also leading to termination.

The analysis of loops differs slightly from that of DAGs in several ways. Firstly, dependences between D nodes all have a weight of 0, which removes the modelling of a finite dispatch width. This is because, in the majority of loops, the latency of the CDS allows all loop instructions to be dispatched in the same number of cycles or fewer, hence dependences between instructions are the bottleneck to performance. In the few cases where dispatch is the bottleneck, this analysis does not limit the issue queue size and hence there is no performance

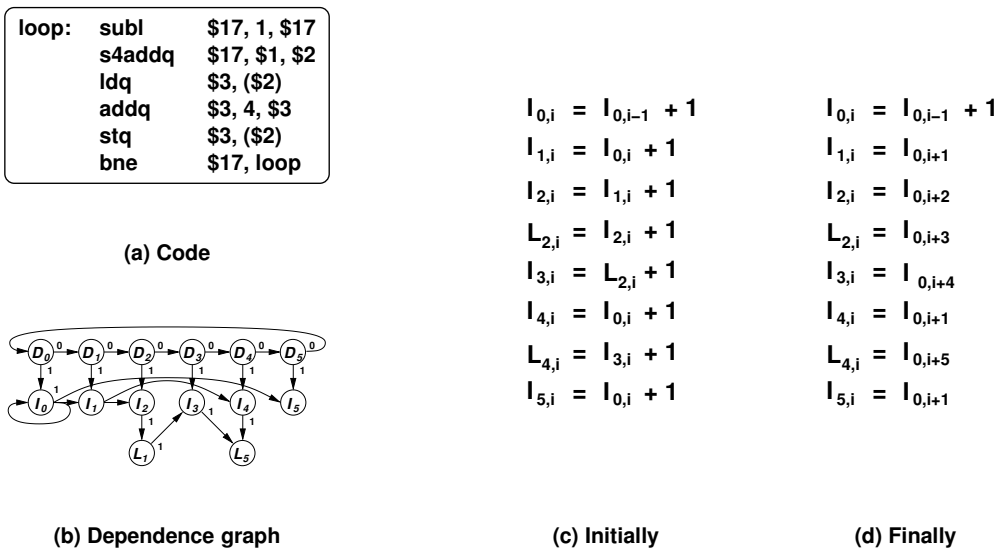


Figure 5.9. Example of loop analysis with equations formed for the dependence graph shown. With a dispatch width of 8 and the issue queue limited to 22 entries, the critical path would not be affected.

loss, but no energy savings result either.

The second aspect of loop analysis that differs from DAG analysis is in the modelling of a finite issue width and functional unit contention. When determining the oldest and youngest instructions to issue together, any contention simply alters the queue size needed accordingly. Hence, if the issue width is eight but nine instructions wish to issue then the calculated size is multiplied by 8/9. This is done in preference to adjusting the equations because a loop with contention executes slightly differently on each iteration and it is difficult to model this effectively.

5.3.4.2 Example

Figure 5.9 shows an example of the compiler analysis for loops. The dependence graph for the loop is shown in figure 5.9(b). In this graph it is easy to see that there is only one candidate cycle for the CDS, containing node I_0 with latency 1. Figure 5.9(c) shows the initial equations formed for the I and L nodes in the loop. Each equation relates the node on loop iteration i with one of its dependence nodes. Where a node has several dependences then equations are usually maintained for each dependence, however only one is shown here for simplicity.

The equation for node I_0 in figure 5.9(c) refers to a previous instance of itself, the instance

of the node on the previous iteration of the loop. The equation for I_0 means that in iteration i, I_0 can issue one cycle after I_0 from the previous iteration $(i-1)$. Continuing on, I_1 can issue 1 cycle after I_0 from the current iteration, I_2 can issue 1 cycle after I_1, and so on.

Once the equations have been formed they are manipulated to remove constants where possible and, hence, determine which nodes issue at the same time, producing the equations shown in figure 5.9(d). Here it is clear that I_1 actually issues at the same time as I_0 from the next loop iteration and I_3 issues at the same time as I_0 from four loop iterations later.

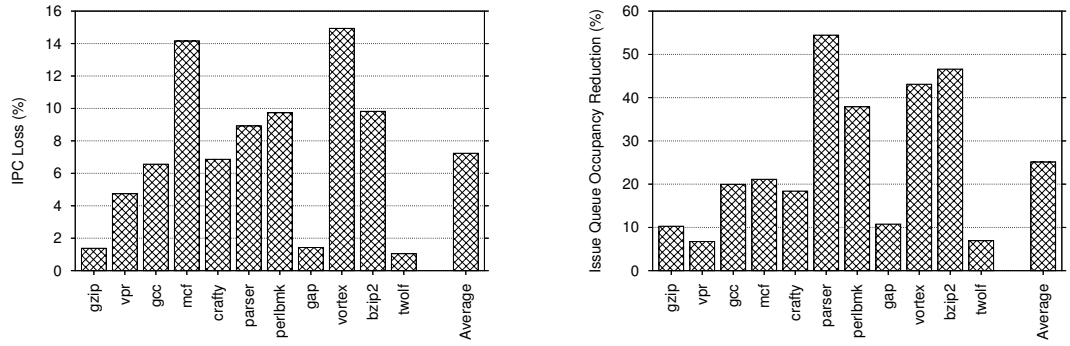
Considering only the I nodes, because they issue from the issue queue whereas the L nodes issue from the load/store queue, it is now trivial to compute the issue queue size needed by the loop. In order that I_0 on iteration $i+4$ can issue at the same time as I_3 on iteration i, they must be in the issue queue at the same time. This would require 22 entries to be available, allowing space for instructions corresponding to I_3, I_4 and I_5 from iteration i, 18 I nodes from iterations $i+1$, $i+2$ and $i+3$ (6 each), and I_0 from loop iteration $i+4$. Providing this many entries would allow parallel execution of this loop without affecting the critical path.

Having performed the compiler analysis, identifying the critical path and determining the issue queue requirements that would not slow it down, the information can be communicated to the processor to perform issue queue throttling. Section 5.4 describes a coarse-grained approach to throttling, where the issue queue requirements are analysed for a whole procedure at a time. Section 5.5 then presents a fine-grained scheme which analyses the requirements for each basic block and throttles only the youngest part of the queue.

5.4 Coarse-grained Throttling

This section describes the first use of the analysis presented in section 5.3 to limit the number of instructions in the issue queue. The critical path analysis is used by the compiler to determine the issue queue requirements for each DAG or loop in its entirety. Limiting of the queue is conveyed to the processor through the use of a special NOOP or tag at the start of each DAG or loop. In addition to this, a special NOOP is also placed after a loop to reset the maximum queue size to the value it was before the loop. This is because the loop may need a different number of queue entries to its surrounding DAG and so allows the queue size to be fit to each structure's requirements.

The rest of this section is structured as follows. Section 5.4.1 describes the trivial microarchitectural changes to the issue queue. Section 5.4.2 then presents the results from using special NOOPs to pass the compiler-inferred queue requirements to the processor. Section



(a) IPC loss **(b) Issue queue occupancy reduction**

Figure 5.10. Performance and issue queue occupancy reductions when limiting using procedure NOOPs for coarse-grained issue queue throttling.

5.4.3 presents the results obtained by tagging instructions, instead of using the special NOOPs.

In each results section, the performance of each scheme is shown in terms of IPC loss normalised to the baseline. The reduction in issue queue occupancy is interesting because this directly affects the energy consumed by the queue due to it being banked. The results for dynamic and static energy reduction are also presented for each approach.

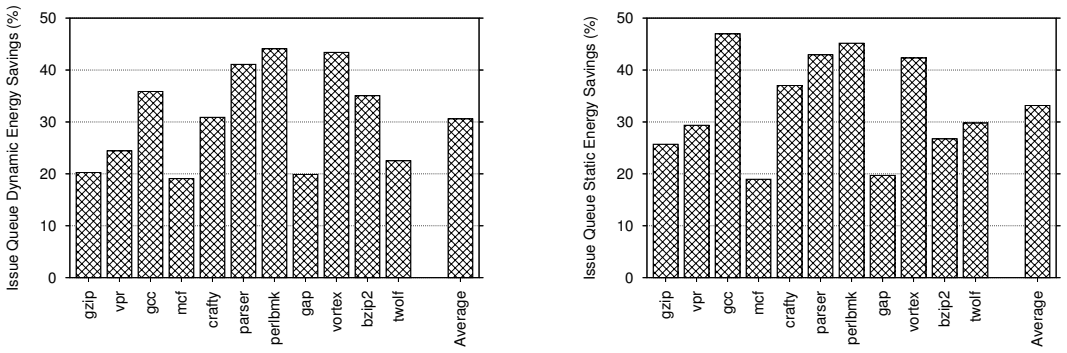
5.4.1 Issue Queue

The changes to the issue queue with this scheme are very minor. The limiting NOOP or tag simply indicates the maximum number of entries, both full or empty, that are allowed between the *head* and *tail* pointers in the issue queue. Instructions cannot dispatch if the *tail* pointer would become more than this allowed distance from the *head*.

In certain situations a limiting NOOP or tag will be encountered that contains a smaller number of entries than is already allowed. To rectify this, enough instructions must be issued from the *head* for the distance between the two pointers to become less than the new maximum distance before dispatching can start again.

5.4.2 Procedure NOOPs

The first evaluation of this scheme was performed using special NOOPs, called procedure NOOPs, to communicate the limiting information. Figure 5.10(a) shows the effect on the performance of each benchmark in terms of IPC loss. Some benchmarks are badly affected,



(a) Dynamic energy savings **(b) Static energy savings**

Figure 5.11. Issue queue energy savings when limiting using procedure NOOPs for coarse-grained issue queue throttling.

such as *mcf* and *vortex* which lose over 14% of their performance. Others experience only a small loss, such as *twolf* which has a 1% drop. On average, the performance loss is 7.2%.

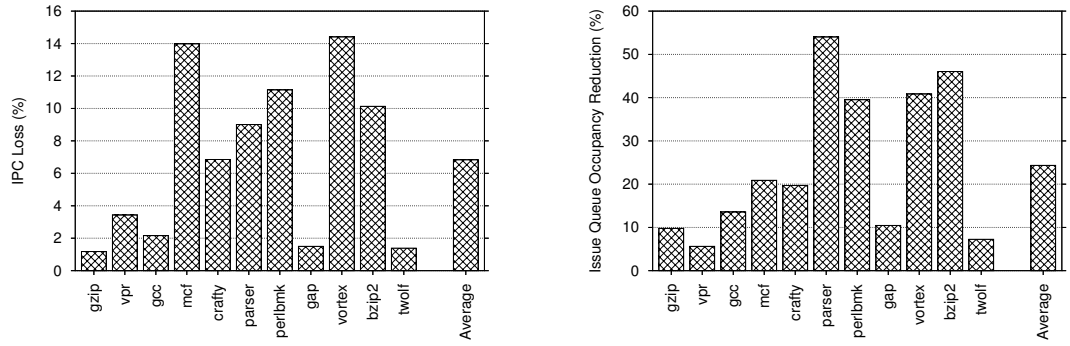
The issue queue occupancy reduction for this scheme is shown in figure 5.10(b). Although benchmarks that experienced a small performance loss, such as *gzip*, *gap* and *twolf*, also experience a small occupancy reduction, the benchmark that benefits the most is *parser* with a 54% drop. This leads to an average occupancy reduction of 25%.

The issue queue energy savings are presented in figure 5.11. Due to the banking of the queue and preventing empty and ready operands from being woken, some benchmarks can achieve around 45% savings in dynamic and static energy, the average reduction being 31% and 33% respectively.

5.4.3 Tags

One side effect of using procedure NOOPs to pass limiting information is that occasionally only seven instructions are dispatched in a cycle, rather than the maximum eight. This is because the NOOPs are not removed from the instruction stream until dispatch, yet are still counted in the number of instructions dispatched even though they do not get put into the issue queue or reorder buffer. To reduce this problem the first instruction in a DAG or loop is tagged with the resizing information, assuming that there were enough redundant bits in the ISA to accommodate the values needed.

The performance loss for each benchmark using these tags is shown in figure 5.12(a). Most benchmarks benefit to some degree from the removal of the NOOPs, *gcc* especially which loses



(a) IPC loss **(b) Issue queue occupancy reduction**

Figure 5.12. Performance and issue queue occupancy reductions when limiting using tags for coarse-grained issue queue throttling.

only 2.2% performance with tags compared with 6.6% with NOOPs. However, in badly performing benchmarks such as *vortex* and *bzip2*, removal of the NOOPs makes little difference.

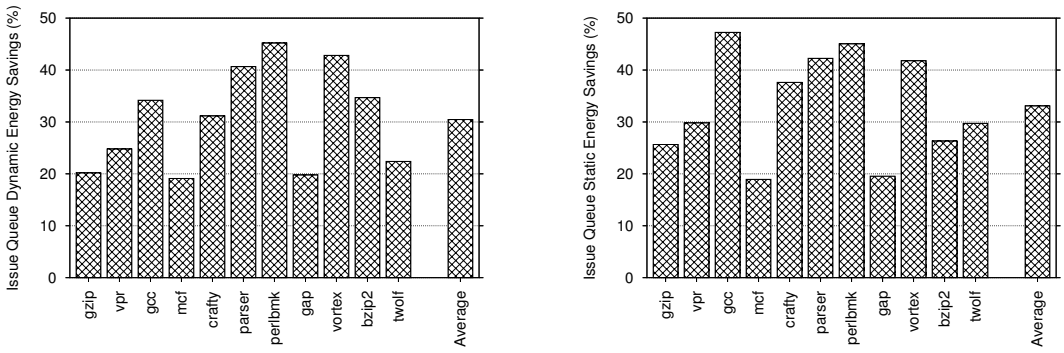
When considering the effects on issue queue occupancy, shown in figure 5.12(b), dynamic energy (figure 5.13(a)) and static energy savings (figure 5.13(b)) there is little change either between different versions of each benchmark or on average. This does show, however, that the NOOPs have little impact on the behaviour of the issue queue but can have a major impact on performance.

In summary, the two schemes presented in this section that perform coarse-grained analysis of a whole DAG's issue queue requirements can be used to reduce the energy consumption of the queue. However, they incur a non-negligible performance loss. The following section attempts to reduce this by performing the throttling at a much finer granularity.

5.5 Fine-grained Throttling

It is clear from section 5.4 that performing issue queue limiting over a whole DAG is too restrictive and creates significant performance losses for some benchmarks. Hence, this section evaluates schemes that perform much more fine-grained throttling.

However, the performance losses are also partly due to the fact that the throttling takes place over the whole queue. When the first instructions in a DAG enter the queue, instructions from a previous DAG will already be there. Without knowing about dependences in the previous DAG, the compiler analysis cannot take into account the issue queue requirements of



(a) Dynamic energy savings **(b) Static energy savings**

Figure 5.13. Issue queue energy savings when limiting using tags for coarse-grained issue queue throttling.

(a) Issue queue on cycle c **(b) Issue queue on cycle c+1**

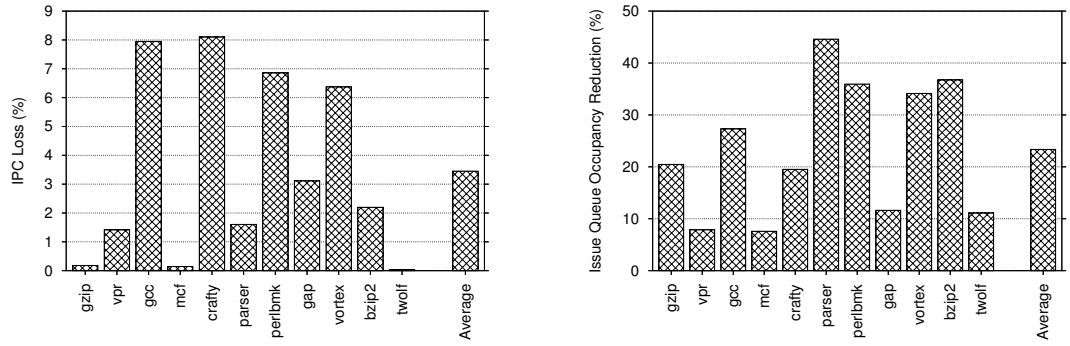
Figure 5.14. Operation of *new_head* pointer with a limit of four entries.

the older instructions. Any assumptions that underestimate these requirements mean that the older instructions are in the queue longer than the compiler realises and stall the dispatch of the new DAG. Hence, this section restricts the throttling of the issue queue to only the youngest part, allowing older instructions to issue without the compiler needing to know about them and consider them during its analysis.

This section is structured as follows. Section 5.5.1 describes the changes to the issue queue required so that only the youngest part of the queue is throttled. Section 5.5.2 presents results obtained using special NOOPs to pass the queue size required by the youngest part of the queue. Section 5.5.3 then presents the same scheme using instruction tags instead of special NOOPs. The final approach is also compared to a state-of-the-art hardware scheme.

5.5.1 Issue Queue

The issue queue requires only minor modifications to allow the new limiting scheme to work. A second *head* pointer, named *new_head*, is introduced which allows compiler control over



(a) IPC loss **(b) Issue queue occupancy reduction**

Figure 5.15. Performance and issue queue occupancy reductions when limiting using block NOOPs for fine-grained issue queue throttling.

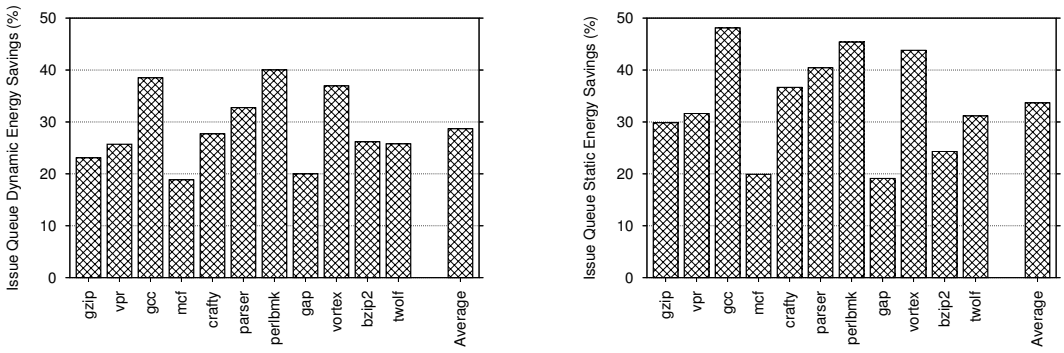
the youngest entries in the queue. The *new_head* pointer points to a filled entry between the *head* and *tail* pointers. It functions exactly the same as the *head* pointer such that when the instruction it points to is issued it moves towards the *tail* until it reaches a non-empty slot, or becomes the *tail*. New instructions being dispatched are still added to the *tail* of the queue.

This scheme is based on the fact that it is relatively easy to determine the future additional requirements of a small program region. Where, in the previous approach, the maximum number of entries between *head* and *tail* were limited, in this section the distance between *new_head* and *tail* is restricted. This is so that instructions from previous basic blocks (and previous DAGs) can be present in the queue without affecting the limiting of the youngest part. So, the instructions between *new_head* and *tail* are from the youngest program region whereas the rest of the instructions (between *head* and *new_head*) are from older ones.

The compiler now sets the maximum number of queue entries allowed between the pointers *new_head* and *tail*. The operation of the *new_head* pointer is demonstrated in figure 5.14. If instruction a issues, the *new_head* pointer moves up to point to the next non-empty instruction, so three slots to d. This means that up to three more instructions can be dispatched to keep the number of entries at four or fewer. So, e, f and g can now dispatch as shown in figure 5.14(b).

5.5.2 Block NOOPs

This section evaluates the new limiting scheme using special NOOPs, called block NOOPs, inserted into the code. The performance of each benchmark is shown in figure 5.15(a). Here it is interesting to see that benchmarks are either hardly affected (*gzip*, *mcf* and *twolf* which lose



(a) Dynamic energy savings **(b) Static energy savings**

Figure 5.16. Issue queue energy savings when limiting using block NOOPs for fine-grained issue queue throttling.

less than 0.2% performance) or experience large performance losses (e.g. *crafty* at over 8%). On average, however, the loss is only 3.5%.

The issue queue occupancy reduction for this scheme is shown in figure 5.15(b). Most benchmarks experience at least a 20% reduction and *parser* gets an occupancy reduction of 45%, the average being a 23% reduction.

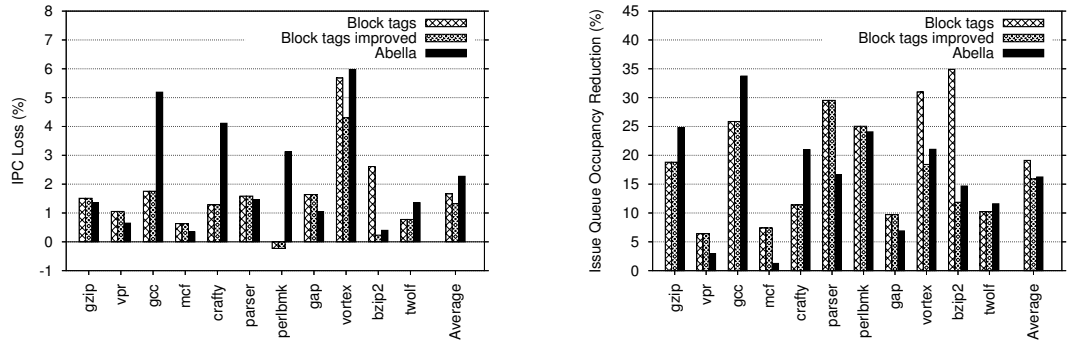
Dynamic and static energy savings are shown in figures 5.16(a) and 5.16(b). The average dynamic and static energy savings are 29% and 34% respectively. The best savings come from *gcc* which experiences 39% dynamic and 48% static energy savings. All benchmarks achieve 19% dynamic energy savings whilst most see their static energy reduced by at least 30% too.

5.5.3 Tags

This section evaluates the use of tags instead of NOOPs to convey the limiting information for the youngest part of the issue queue. In each graph three schemes are shown. The first is the approach that simply uses tags instead of NOOPs to pass limiting information to the processor. This scheme is called *Block tags*.

The second scheme, called *Block tags improved*, is derived from the *Block tags* technique. By hand, limited inter-procedural analysis was applied to *vortex* and *bzip2* to reduce the functional unit contention across procedure boundaries for the most heavily used procedures. This would typically be available in a mature industrial compiler but is currently absent in the SUIF prototype.

The final scheme was implemented for comparison with a state-of-the-art hardware ap-



(a) IPC loss **(b) Issue queue occupancy reduction**

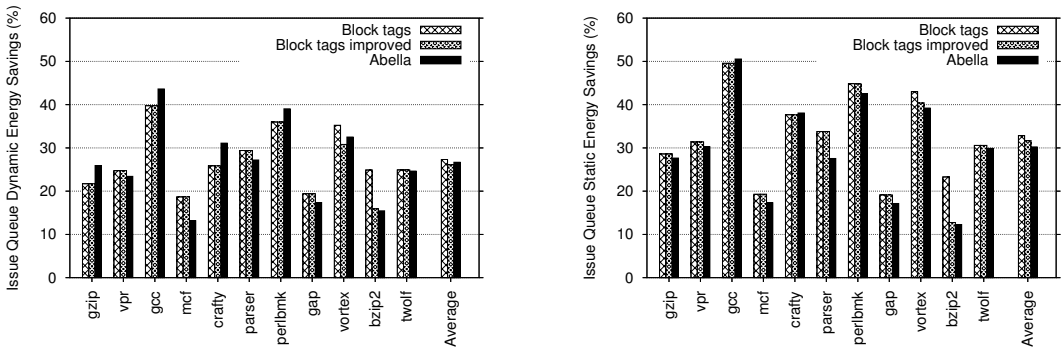
Figure 5.17. Performance and issue queue occupancy reductions when limiting using tags for fine-grained issue queue throttling.

proach. It is taken from papers published by Abella and González [5, 1] and implemented within the same simulation infrastructure as all other experiments. The results for *Block tags* and *Block tags improved* are compared to their *IqRob64* scheme as this gave the most energy savings and is henceforth referred to as *Abella*.

As can be seen from figure 5.17(a), the compiler schemes lose less performance than *Abella*, and *perlbmk* even gains slightly (0.2%). All approaches perform badly on *vortex* but *Abella* cannot cope well with *gcc*, *crafty* or *perlbmk*. The inter-procedural analysis performed in *Block tags improved* considerably reduces the performance loss of *bzip2* to just 0.2%. On average *Block tags* loses 1.7%, *Block tags improved* 1.3% and *Abella* 2.3% performance.

The issue queue occupancy reduction, in figure 5.17(b) shows wide variations between benchmarks. The effect of adding inter-procedural analysis to *vortex* and *bzip2* is that they get less of an occupancy reduction. In general, a high issue queue occupancy reduction translates into high dynamic and static energy savings (figures 5.18(a) and 5.18(b)). The static energy of the issue queue is completely dependent on the number of banks that are on each cycle whereas the dynamic energy consumption is dependent on the number of instructions waking others and reads and writes to the queue, as well as the occupancy. The average dynamic energy savings of the *Block tags* scheme is the same as *Abella* (27%) whereas it is reduced slightly to 26% in *Block tags improved*. The static energy reduction is, on average, better than *Abella* in both compiler schemes. The *Block tags* approach reduces it by 33%, *Block tags improved* by 32% and *Abella* by 30%.

Limiting the issue queue in this manner reduces the number of instructions dispatched by



(a) Dynamic energy savings **(b) Static energy savings**

Figure 5.18. Issue queue energy savings when limiting using tags for fine-grained issue queue throttling.

5% (13% in the case of *perlbmk*). This, in turn, reduces register pressure meaning that the banked register file has more opportunities for turning off banks and saving both static and dynamic energy.

Figure 5.19 shows the reduction in integer register file occupancy. There is a slight reduction in all benchmarks with the average being 8.5% for *Block tags* and 7.0% for *Block tags improved*. This leads to the energy savings shown in figure 5.20(a) for dynamic energy and figure 5.20(b) for static energy. The *Abella* scheme manages to save more dynamic and static energy, but suffers a higher IPC loss. On average, *Block tags* gets 12.4% dynamic and 8.2% static energy savings whilst *Block tags improved* gets drops of 12.0% and 8.1% in its dynamic and static energies respectively.

5.6 Summary

This chapter has presented novel techniques to dynamically resize the issue queue using the compiler for support. The compiler analyses and determines the number of issue queue entries needed by each program region and encodes this number in a special NOOP or a tag with the instruction. The number is extracted at dispatch and used to limit the number of instructions in the queue. This has the effect of reducing the issue queue occupancy and thus the amount of energy dissipated.

Results from the implementation and evaluation of the proposed schemes show 31% dynamic energy savings with a 7.2% average IPC loss for a basic scheme which attempts to

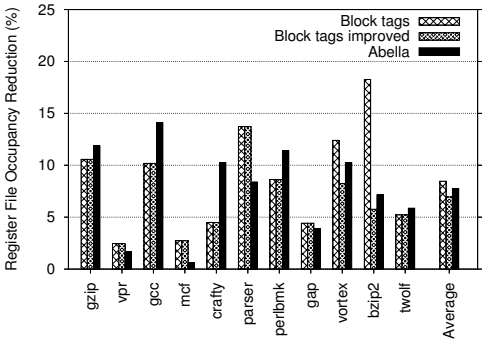
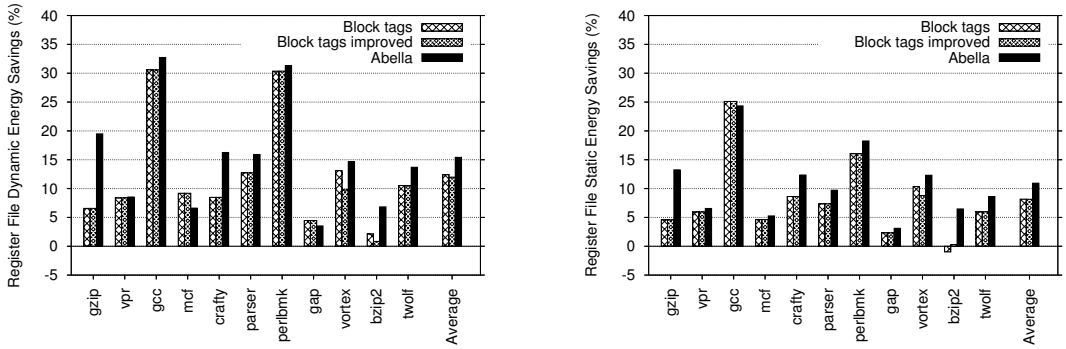


Figure 5.19. Register file occupancy reduction when limiting using tags for fine-grained issue queue throttling.



(a) Dynamic energy savings

(b) Static energy savings

Figure 5.20. Register file energy savings when limiting using tags for fine-grained issue queue throttling.

determine the whole queue size needed. By determining the requirements of only the youngest part of the queue, the performance loss can be reduced to just 3.5% when using the *Block NOOPs* scheme to convey the information. Tagging instructions and using improved analysis reduces this further to just 1.3%, compared with 2.3% for a state-of-the-art hardware scheme by Abella and González [5]. Both compiler and hardware schemes save similar amounts of static and dynamic energy in the issue queue.

A side effect of limiting the number of instructions in the issue queue is that fewer registers are needed and thus dynamic energy savings of 12.0% and static energy savings of 8.1% are also achieved in the integer register file. Therefore, the compiler-based technique has a smaller performance loss whilst saving similar amounts of energy as a state-of-the-art approach and needing less complex hardware.

Chapter 6

Register File Energy Savings

The register file in a modern superscalar processor facilitates out-of-order execution by eliminating false (WAR and WAW) dependences between instructions. However, it is one of the most energy-consuming structures within the processor [25] with a high latency and is a hotspot whose cooling system cost in future processors will increase non-linearly compared to the amount of heat removed [36].

Previous research has noted that registers are idle for many cycles after their last use, before being placed on the free-list to be assigned to a new instruction [17, 62]. This is because a register cannot be released until the instruction redefining its logical register commits, in order to maintain a precise processor state in the event of an exception or interrupt.

Early register releasing has been proposed [64, 58, 62, 27] to remove this idle time by releasing a register before the commit of its redefining instruction. This allows the register to be reused by a newly dispatching instruction and reduces the overall occupancy of the register file, meaning that a smaller register file can be used without a drop in performance. Hardware early releasing schemes suffer from the fact that, without speculative releasing, they can only release a register early when the redefining instruction enters the pipeline. This is to ensure that no future instructions will need to read the value in the register that is being released. However, it may be many cycles after a register's last use before the redefining instruction is dispatched, during which time the physical register is not accessed and an opportunity for early releasing is missed.

This chapter shows that an idealistic hardware oracle with full knowledge of the last use of each register can significantly reduce register file occupancy, which directly affects the amount of energy consumed. However, existing, complex hardware schemes are only able to achieve

between 11% and 23% of this reduction. The reason for this poor performance is that the hardware, in practice, does not know the last use of any register and has to be conservative in its release policy. This chapter considers the design space of compiler supported approaches and shows that they are able to provide dramatic improvements. The compiler has global knowledge of the program and can guarantee when a register value will not be used again.

This chapter shows that a simple compiler-assisted scheme, with minimal hardware support and no ISA modifications, can achieve 81% of the hardware oracle's occupancy reductions. A more aggressive implementation can achieve 83% of the maximum potential benefit. A scheme with tagged instructions which achieves 89% of the benefit of a hardware oracle is also developed. Finally, an evaluation of the best techniques is made across varying register file sizes, which shows that they provide large occupancy reductions and significant IPC gains, particularly for small register files, outperforming the best current techniques.

The rest of this chapter is structured as follows. Section 6.1 motivates the use of early releasing to reduce register pressure and the ability of the compiler to obtain the majority of the benefits available. Section 6.2 describes the changes to the microarchitecture that are common to all subsequent schemes. Sections 6.3 and 6.4 describe commit and issue-based early releasing approaches whilst section 6.5 combines the best schemes from each. Section 6.6 evaluates compiler and hardware techniques across varying register file sizes and finally section 6.7 summarises the main points of this chapter.

6.1 Motivation

This section describes early releasing, motivating its use in reducing register pressure. It also evaluates a hardware oracle and shows that a compiler-directed releasing scheme can achieve almost all the potential benefits. Two situations where registers can be released early are considered: in the issue stage of the pipeline or in the commit stage.

6.1.1 Early Register Releasing

Early register releasing can be used to reduce the number of physical registers occupied at any one time i.e. register pressure. This allows unused physical registers to be turned off which, using the banked register file described in chapter 2, saves energy. This technique can also be used to design processors with smaller register files without affecting performance.

To illustrate how the proposed approach works, consider the example in figure 6.1 where,

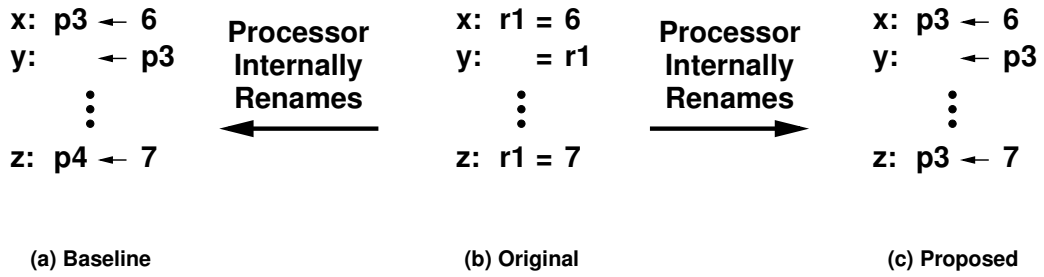


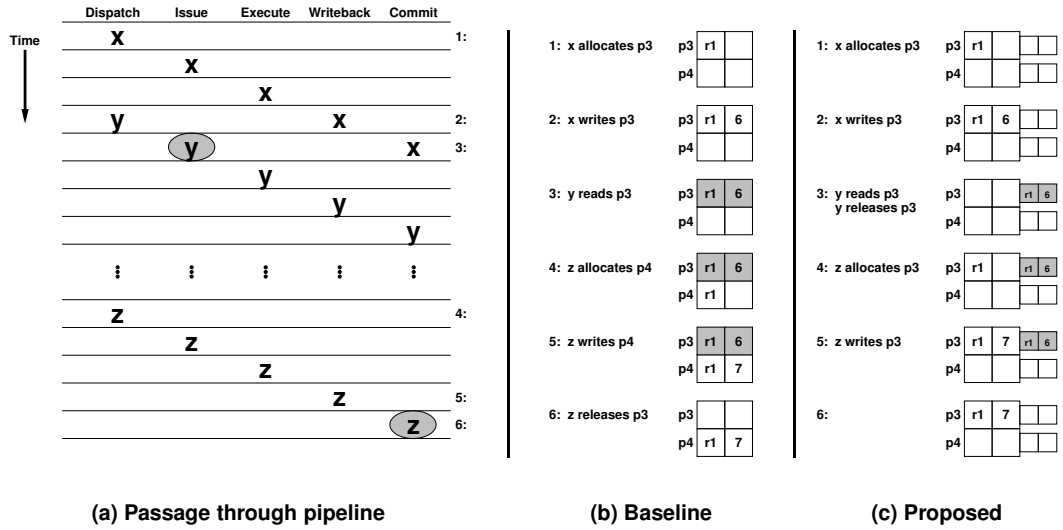
Figure 6.1. Two registers are used when the baseline internally renames registers. With early releasing only one is needed.

to aid readability, reads and writes to registers are shown using pseudo-code. This example shows a simple assembly code fragment (figure 6.1(b)) where the value 6 is written into register r1 in instruction x, read in instruction y and then a new value (7) is assigned to register r1 in instruction z some time later. In the normal baseline scheme (figure 6.1(a)) the assembly program register is allocated by the processor at runtime to a physical hardware register, say p3, and the value 6 written to it by instruction x and read in by instruction y as before. When r1 is written to again, the baseline assigns a new unoccupied or unallocated physical register to r1, say p4, so as to eliminate the anti-dependence from y to z and the output-dependence from x to z, the intention being to prevent false storage-based dependences which may slow the program down. So, if instructions x or y were delayed for any reason (e.g. waiting for another operand or functional unit contention), the out-of-order superscalar hardware could continue to execute z and later dependent instructions without stalling. However, if at runtime the read of r1 occurs before the write in instruction z, then there is no need to allocate a new physical register and the same physical register p3 could be reused (figure 6.1(c)).

The passage of these instructions through a pipeline is shown in figure 6.2. The state of the registers at key points is also considered. Section 6.1.1.1 describes the events occurring in the baseline processor, whereas section 6.1.1.2 explains the same events using an early releasing technique proposed in this chapter. Section 6.1.1.3 summarises other early releasing schemes.

6.1.1.1 Baseline Case

The diagram in figure 6.2(a) shows the passage of the instructions through the pipeline whilst 6.2(b) shows the state of the baseline physical register file. A five stage pipeline is shown to aid presentation. Physical registers are allocated when a defining instruction dispatches at the



(a) Passage through pipeline **(b) Baseline** **(c) Proposed**

Figure 6.2. Passage of three instructions through the pipeline and the state of the register file. Normally p3 is not released until z commits. With early releasing this can be done much earlier, when y issues, as this is the last use of p3. Other hardware techniques must wait for z to dispatch to be certain that r1 will not be used again.

beginning of the pipeline and are released at the time of the redefining instruction's commit. So, at point 1, p3 is allocated to logical register r1[1]. At point 2 the value (6) produced by x is written into the physical register p3. At point 3 it is read for the last time (denoted by the shaded box) as y issues. At point 4, possibly much later, z dispatches and so register p4 is allocated. The value (7) generated by z is written into the physical register p4 at point 5. Finally, at point 6 in the baseline case, the register holding the previous version of r1 (p3) is released. The physical register p3 needlessly retains the value of r1 from point 3 to point 6 in the baseline case.

6.1.1.2 Proposed Scheme

When early releasing is used (figure 6.2(c)) the same events happen at points 1 and 2, i.e. register p3 is allocated to r1 and then the value (6) is written in at the writeback stage. However, at point 3, once instruction y has read p3, it is immediately released allowing it to be reused later as it is known that there are no other consumers of this value.

[1]The logical register that each physical register corresponds to is not actually kept in the register file itself but is merely shown for clarity.

Although the value will no longer normally be used, rather than discarding it, it is copied into a cheap backup storage using the checkpointed register file proposed by Ergin *et al*. [27] and summarised in section 6.2.1. So, if there were an exception or mis-speculation between instruction y and z the value of register r1 could be retrieved from the slow backup store. The cost of recovering from a mis-speculation by flushing the pipeline or by handling an exception completely amortises any cost due to recovering the old value of r1. The checkpointed register file is discussed in more detail in section 6.2.1 and the handling of interrupts and exceptions in section 6.2.3.

Continuing with the example, at point 4, the register is allocated to the now free physical register p3, which is written to at point 5 with the value 7. At point 6 the old value of r1 will never be needed as its new value has committed so the backup copy just needs to be cleared. In reality this does not involve anything more complicated than marking it invalid.

Thus this scheme is able to release registers early and guarantee correct behaviour in the case of mis-speculations or exception handling with the support of a small amount of cheap backup storage. If p3 and p4 were in different register banks, p4 could be gated off for the entire instruction sequence, saving static and dynamic energy.

6.1.1.3 Other Schemes

Other early releasing schemes would release p3 later in the pipeline. Ergin *et al*. [27] release when the redefining instruction (z in this example) has entered the pipeline, the original defining instruction has committed and all consumers have read the value. This is at point 4 in the diagram. Monreal *et al*. [62] release when the redefining instruction becomes non-speculative and all consumers have read the value. This would occur somewhere between points 4 and 6 in the diagram. Martin *et al*. [58] release at the earliest at the commit stage of the last user (y in this example). However, their scheme would usually release later, at the commit of an explicit early release instruction. This could be any time between the commit of the last user and the point at which the baseline releases.

6.1.2 Hardware Oracle versus Compiler Analysis

Having motivated the use of early releasing to reduce register pressure, this section considers the extent to which this can be achieved and the ability of the compiler to help. It also introduces the notions of commit-based and issue-based early releasing.

Consider the situation where the processor has knowledge of the future, i.e. an oracle.

Specifically, this oracle knows the instructions that are the final consumers of each register. This is gained by executing each benchmark twice, generating a trace the first time which can then be consulted the second to determine the last consumer information. Using this information the processor can release a register far earlier than usual without having to wait for the dispatch of a redefining instruction. This oracle, although unrealistic in practice, serves as the lower bound on the register file occupancy the processor alone can achieve.

Registers can be early released either at the commit of a certain instruction, or when it issues. The former is referred to as a commit-based scheme, whereas the latter is known as an issue-based approach. The potential benefits to be gained from both commit-based and issue-based early releasing schemes are evaluated using this oracle.

A second idealistic case considered is where the compiler is able to pass full information about the last use of each register to the hardware without considering the hardware cost and ISA impact. An instruction is tagged if it is the last consumer of one of its source registers, which allows the processor to release it early. Additionally, at the start of each basic block, tags are placed for each register that is live out of a predecessor but not live into the current block. The compiler analysis considers all control flow paths through a program, including loops, and incorporates inter-procedural analysis to mark the last use of each register. Again, commit-based and issue-based schemes are evaluated.

Figure 6.3 shows graphs of the IPC and register file occupancy (based on the architecture described in chapter 4) for the hardware oracle and compiler analysis. As figure 6.3(a) shows, there is little change in the IPC of the benchmarks when early releasing occurs. However, as figure 6.3(b) shows (and figure 6.3(c) shows more clearly), there can be a large reduction in the register file occupancy. More savings can be achieved by the hardware oracle if registers are released in the issue stage of the pipeline (from 85 down to 55), when all consumer instructions have read the data, rather than in the commit stage (when it is reduced to 68). This is because, after execution, instructions can wait many cycles in the reorder buffer before committing.

Figure 6.3(d) shows the average occupancy of the oracle, compiler analysis and three state-of-the-art approaches in terms of the percentage of the total reduction possible (85 down to 55), they have achieved. So, the baseline, by definition, has a 0% reduction (not shown) while the hardware oracle with an issue based releasing scheme, by definition, achieves 100% of the possible reduction in register occupancy (again not shown). The existing approaches of Monreal *et al*. [62], labelled *Monreal* in figure 6.3(d), Ergin *et al*. [27], labelled *Ergin*, and Martin *et al*., labelled *Martin*, do not significantly reduce the occupancy. *Monreal* achieves a

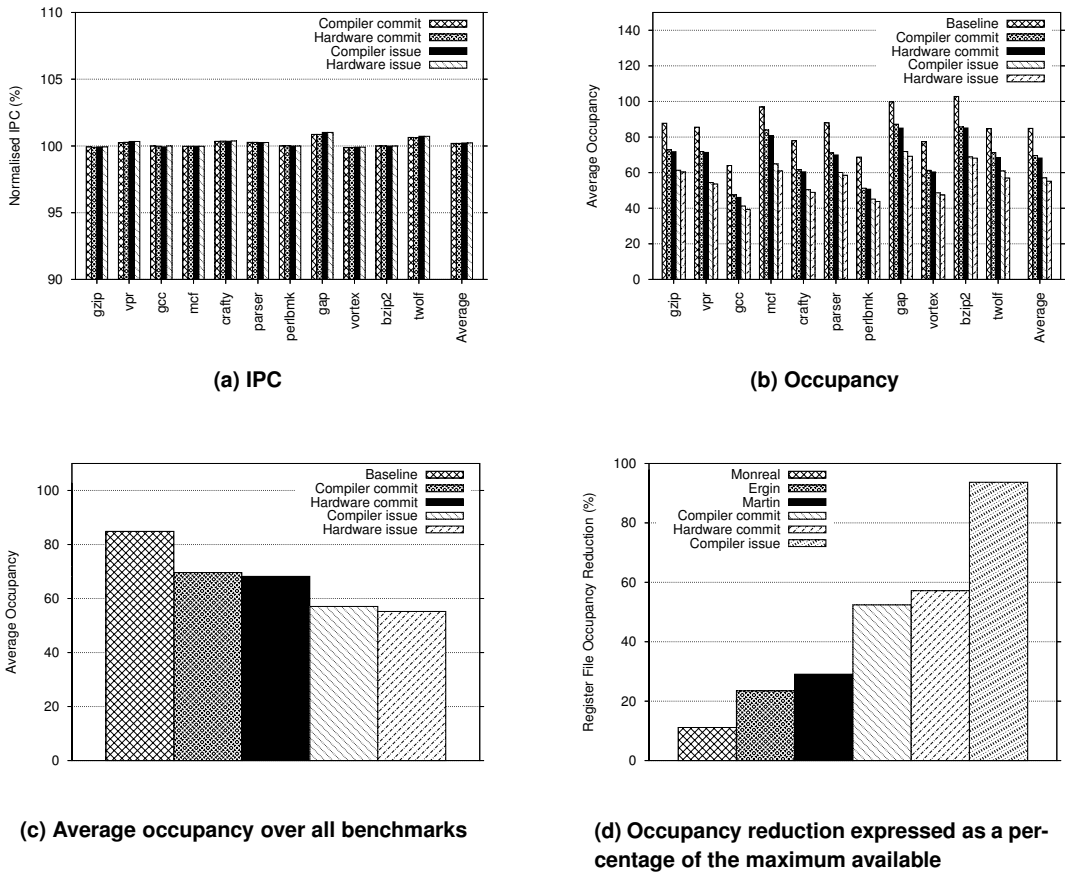


Figure 6.3. The hardware oracle and best compiler analysis with early releasing at issue and commit.

11% reduction of the maximum possible (from 85 down to 82), *Ergin* achieves 23% (to 78) and *Martin* achieves 29% (to 76). This compares to the hardware oracle releasing at commit which achieves a 57% reduction of the maximum available (to 68).

What is immediately obvious is that the compiler-based scheme achieves almost the same savings as the hardware oracle i.e. 57% reduction for commit based releasing and a 94% reduction for issue based releasing. This shows that exploring realistic compiler-based implementations of early releasing is potentially worthwhile. The remainder of this chapter attempts to determine realistic compiler-based techniques, exploring the trade-off between microarchitecture modification and register file occupancy reduction.

6.2 Microarchitecture

This section briefly describes the changes made to the register file and associated rename logic that are used throughout this chapter.

Unlike chapter 5, this chapter uses issue-bound operand fetching because the majority of the literature concerned with register file optimisations assumes this. In particular, sections 6.1.2 and 6.6 compare the proposed schemes with hardware techniques that use this type of operand fetching and therefore so does this chapter, to allow meaningful comparisons to be drawn. However, the schemes presented in this chapter could easily be applied to a dispatch-bound fetching architecture, creating opportunities to release registers even earlier.

As in chapter 5 and explained in chapter 2, all banks in the baseline simulator's register file are on permanently.

6.2.1 Checkpointed Register File

The checkpointed register file was proposed by Ergin *et al*. to aid their early releasing scheme [27]. It can hold a copy of each register that is released early in cheap backup storage that can be recovered easily and quickly. This enables the processor to maintain a consistent state and thus implement precise interrupts and exceptions and recover from branch mis-predictions. An example of a single bit in this register file is shown in figure 6.4.

To implement backup storage an extra bitcell (called the shadow cell) is connected to the main bitcell. Two extra wires are needed to signal a store from the main cell's value into the shadow bit (a *Checkpoint* line) or to copy from the shadow cell back into the main bitcell (*Recover*).

The area overhead of the shadow bitcells is independent of the number of ports. For the register file used in this thesis with 16 read and 8 write ports, the area overhead is 19.4% [27]. The delay overhead is less than 0.5% since no extra gate capacitance is added to the lines [27]. The extra width and height of the checkpointed bitcell increases the wordline and bitline energy consumption, but this affects the energy dissipated in a read or write by only a small amount. There is also a small amount of energy consumed when checkpointing and recovering data from the shadow cells. All additional energy consumption is accounted for in the experiments performed.

In order to keep the additional static power dissipation to a minimum, a super-drowsy circuit is employed for the shadow bitcells [44]. When turned on, the supply voltage arrives through a wide-channel transistor, but when off, a long-channel transistor supplies a lower

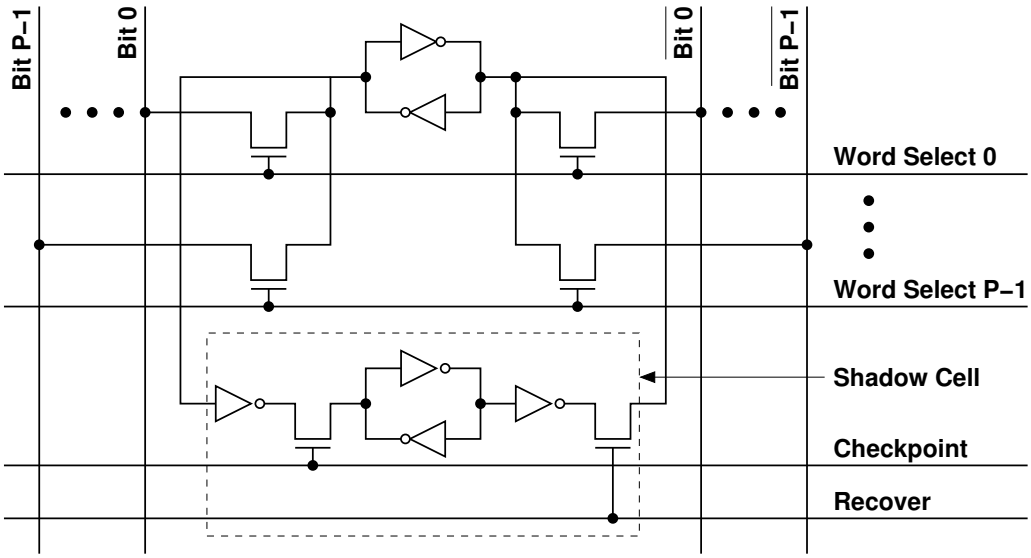


Figure 6.4. A checkpointed bitcell with *P* ports [27].

supply voltage to preserve the state of the bitcell. With a drowsy voltage of 250mV the leakage energy of the circuit can be reduced by 98% [44]. This technique is only applied to the checkpointed bits where a fast access time is not needed.

6.2.2 Early Releasing

Each register needs to keep track of whether the value held in its shadow bitcells is valid. A single bit is added to each register, called the *checkpointed* bit, which indicates that this value is needed.

The register retirement map table needs to keep track of each logical register, whether it is held in the main or shadow bitcells of the physical register pointed to. A *checkpointed* bit is added to the register retirement map table (one per logical register) which, when set, indicates the required value can be found in the shadow bitcells of its physical register.

When an instruction commits, it releases the previous version of its logical destination register. The register retirement map table is consulted and the relevant *checkpointed* bit read. If it indicates the actual register is held in shadow bitcells, then the only tasks that need performing are to clear the *checkpointed* bits in the map table and the physical register pointed to. If the *checkpointed* bit shows that the actual register is kept in a main bitcells, then the previous

version of the logical register is released in the normal way.

The total overhead of using the checkpointed register file for early releasing is 32 bits in the register retirement map table (one per logical register) and 112 bits in the integer register file (one per physical register). Certain schemes presented later require extra bits to the reorder buffer and map tables which are described in detail later.

Only integer programs are simulated (see section 4.1) so the floating point register file is not considered for optimisations. Henceforth in this chapter, references to the register file always mean the integer register file.

6.2.3 Interrupts and Exceptions

When an interrupt or exception occurs, the pipeline is emptied. Before the interrupt or exception handler can be invoked, all logical registers must be represented as non-checkpointed physical registers in order to maintain a precise processor state. At this time there may be some registers that are checkpointed and of these, some may have other valid values in their main register cells. These need to be moved so that the checkpointed values can be safely restored, without overwriting the values already there.

To achieve this, the processor consults the retirement map table to determine registers that are blocking checkpointed values in the main bitcells. It then issues a `MOV` instruction for each one, placing them in different physical registers. It is guaranteed that there will always be a free physical register to move the checkpointed values into because there are more physical registers than logical. Once this has occurred, the checkpointed values can be safely restored before executing the interrupt or exception handler. This scheme ensures no registers are held in shadow bitcells when control passes over. Although this may take several cycles longer than in the baseline case, the infrequent nature of interrupts and exceptions, compared to the savings gained from these schemes, make this worthwhile.

6.3 Commit Releasing

This section describes and evaluates commit-based early releasing techniques. These schemes release registers after the commit of the instructions performing the release. These approaches are attractive as the microarchitecture requirements are relatively modest. Four schemes are evaluated, with varying impact on the ISA. These are: using special NOOPs, which have no impact on the ISA; releasing at branches; releasing at procedure boundaries; and releasing

through tagging the last use of each instruction. This section describes the minimal architectural impact of each of the schemes and provides a short description of the compiler analysis and ISA modifications required.

6.3.1 Commit NOOPs

To implement early releasing techniques, ideally there should be as little impact on the processor and ISA as possible. One approach is to use a special NOOP, called a commit NOOP. This is simple to implement, requires no changes to the ISA and has no effect on a program's semantics. Each NOOP can hold an encoding of the registers that are to be released, allowing the processor to release them early at the commit stage.

6.3.1.1 Microarchitecture Changes and ISA Impact

The commit NOOPs need to be dispatched in the same manner as any other instruction. They are stored in the reorder buffer, along with the other instructions, so that they can release a set of registers upon commit. However, they are not executed since they perform no operation. When a commit NOOP is committed, the processor attempts to release a set of registers early. To represent all 32 logical registers in the processor, only 5 bits are needed. Each NOOP has 25 bits free, allowing it to release a maximum of 5 registers at commit. However, in the experiments, on average, each one released just 3 registers.

For each encoded register, the retirement map table is accessed, the correct physical register found and the relevant *checkpointed* bit read. If the *checkpointed* bit for a register is clear then, in the following cycle, it can be released early. If the *checkpointed* bit is set then the value held in the shadow bitcells of the physical register is needed, so the register cannot be released early. Each register that is released early is recorded in the register retirement map table by setting the *checkpointed* bit, as described in section 6.2.2. The modest overhead of this technique is enough ports to access the *checkpointed* bits in the physical registers, via the register retirement map table, and check that they are set. By allowing two commit NOOPs to commit each cycle, this means that there needs to be ten extra ports to the *checkpointed* bits.

6.3.1.2 Compiler Analysis

Determining the set of registers that will be released early by each commit NOOP involves constructing the control flow graph and liveness information for each procedure. Commit NOOPs

are inserted at the start of each basic block to release registers that have been used along all paths before it, but are not live into it.

Liveness information is constructed for a control flow graph using the sets of registers that are written and read by each instruction. Each node, N, in the control flow graph is a basic block in this analysis. The set of variables live immediately before node N is defined as $liveIn[N]$ and $liveOut[N]$ is the set of variables live immediately after N. The set $read[N]$ is the set of registers that are used before being defined in node N and $write[N]$ is the set of registers that are defined by N. Two further sets, $preds[N]$ and $succs[N]$, the sets of predecessors and successors of N respectively, can be obtained from the control flow graph. The following equations relate the sets:

$$liveIn[N] = read[N] \cup (liveOut[N] - write[N])$$

$$liveOut[N] = \bigcup_{\forall S \in succs[N]} liveIn[S]$$

Initially, $liveIn[N] = \emptyset$ for each node, N, and a post-order traversal of the control flow graph is performed, to create the *liveIn* and *liveOut* sets for each node. This is iterated in a work-list algorithm until the equations reach a fixed-point.

Using this information, along with $used[N]$ (the set of registers used at any point in N) commit NOOPs are inserted containing register R at the start of each node, providing the following condition holds:

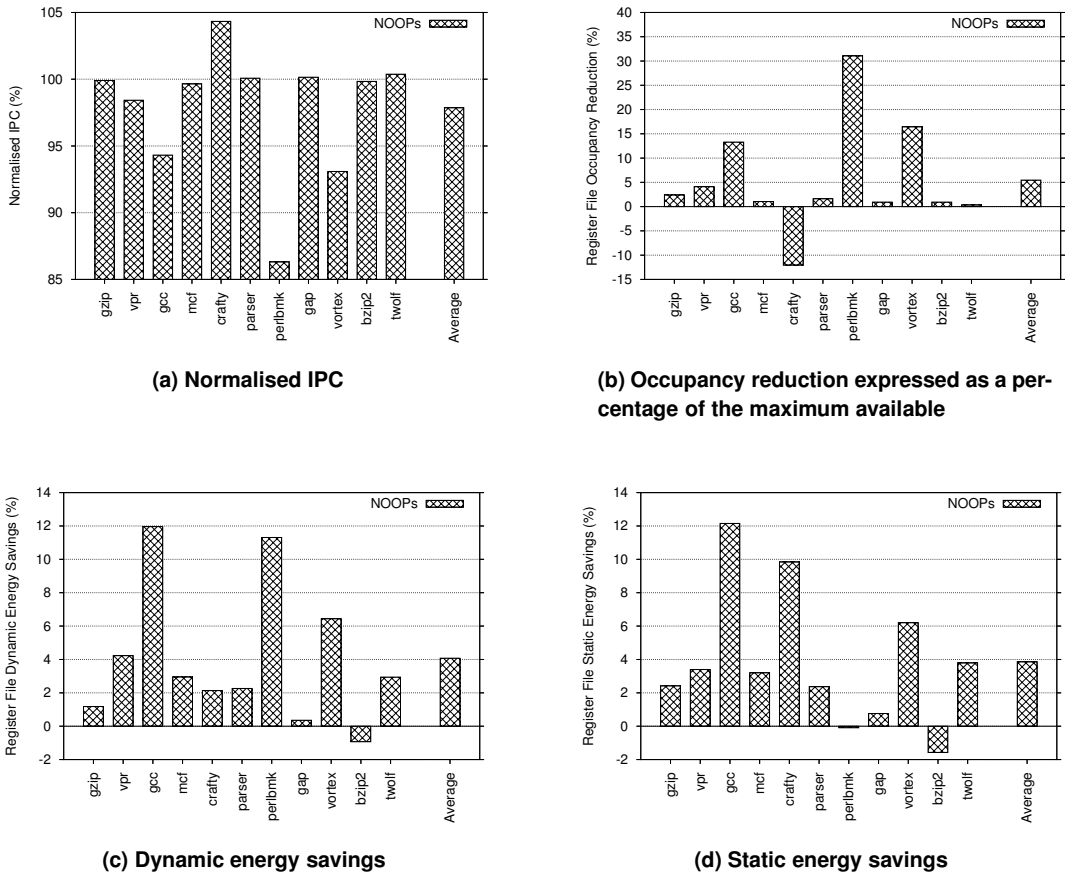
$$R \notin liveIn[N] \wedge \forall P \in preds[N] : R \in used[P]$$

In other words, R must be used in all N's predecessor nodes but cannot be live immediately before N.

6.3.1.3 Results

Figure 6.5 shows the effect of releasing registers early through this technique. In this chapter, most schemes produce slight performance gains, as opposed to chapter 5 where IPC losses are common. Therefore, performance graphs show the IPC normalised to the baseline. Occupancy reduction and energy saving graphs are shown as before.

As figure 6.5(a) shows, some benchmarks experience an increase in IPC due to reduced register pressure. However most experience a decrease due to the commit NOOPs taking up resources within the processor (e.g. the reorder buffer). For example, *perlbmk* has its performance reduced from an IPC of 1.7 to 1.5 with commit NOOPs. Register file occupancy is



(a) Normalised IPC

(b) Occupancy reduction expressed as a percentage of the maximum available

(c) Dynamic energy savings

(d) Static energy savings

Figure 6.5. Releasing using commit NOOPs.

normalised to be the difference between the baseline (at 0%) and the hardware oracle (at 100%) in figure 6.5(b), as shown before in figure 6.3(d). The average register file occupancy is slightly reduced overall, bringing savings of 5% of the maximum possible. This translates into average dynamic energy savings of 4% and average static energy savings of 4%, with *gcc* benefiting the most.

Although this technique shows some register file occupancy reduction and energy savings, it loses performance and there remains significant room for improvement. Therefore, this approach is not considered further.

6.3.2 Branches

As discussed in section 6.3.1.3, the downside of using commit NOOPs is that they take up processor resources. This scheme considers the case where branch instructions have spare

bits that can be used to release a set of registers early, in much the same way as the commit NOOPs. The differences are that the branch instruction is actually executed and that it occurs at the end of a basic block, rather than at the start, so it should release registers used in the block it belongs to. Using branches also means that some blocks will not have their registers released early because some do not terminate with a branch instruction.

The microarchitecture changes required to support releasing on branches are exactly the same as for commit NOOPs, using just the *checkpointed* bits of the physical registers and the register retirement map table. Experiments find that in most cases only 1 register, requiring 5 bits, is released by each branch instruction.

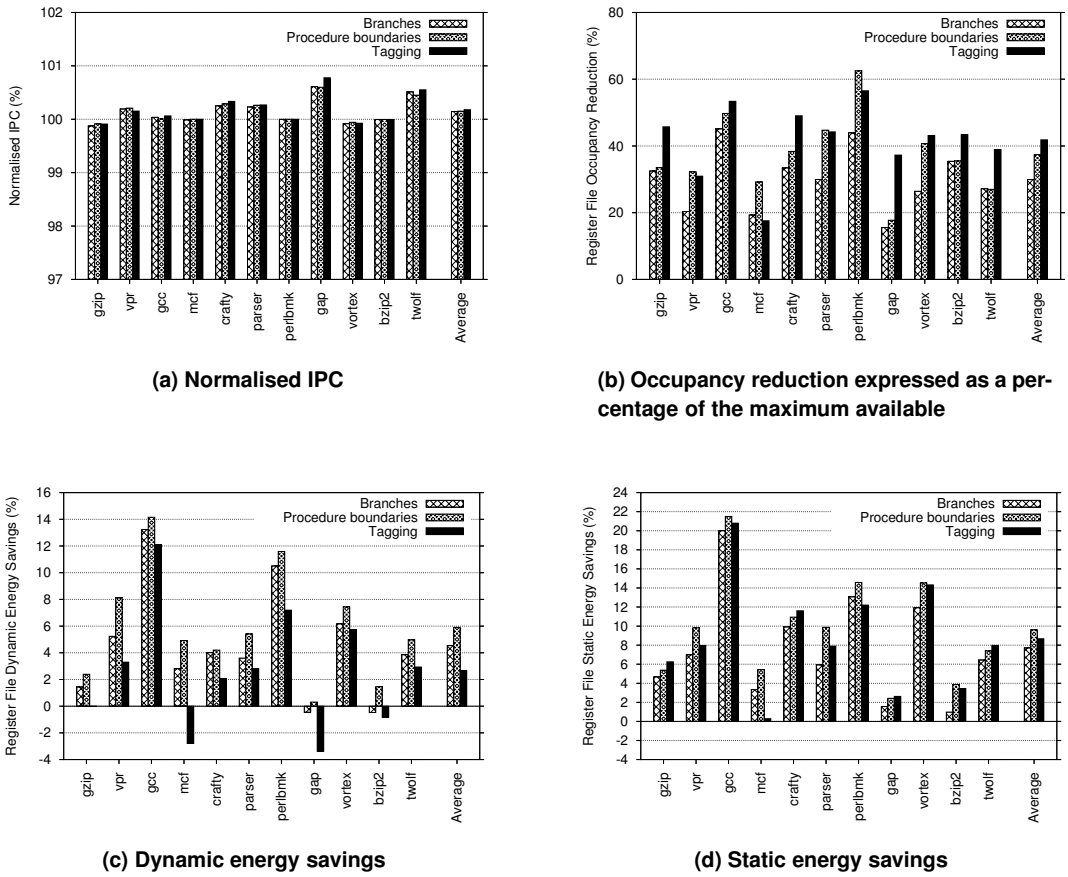
The compiler analysis for this scheme is similar to that used when releasing with commit NOOPs, using liveness information about each node in the control flow graph. The only difference is the final condition that determines whether a register will be released by a branch. Register R can be released by the branch at the end of node N if the following condition holds:

$$R \in used[N] \wedge R \notin liveOut[N]$$

In other words, R must be used in N but must not be used in any node following N before being redefined.

6.3.3 Procedure Boundaries

Another approach to early releasing is to release caller-saved registers at calls and returns, if they are guaranteed not to be live afterwards. The exact set can be altered, depending on the calling conventions the compiler wishes to use, and can be passed to the processor by a special NOOP, called a caller-saved NOOP and different to that mentioned in section 6.3.1, which encodes the set of registers the compiler will use as caller-saved. It is placed before entry to foreign code, to indicate that no further releasing of this type should occur. Another is placed after the corresponding return to restart early releasing again. Whenever the compiler alters its set of caller-saved registers, it can place a caller-saved NOOP to inform the processor of the change immediately beforehand. The set of registers to release can be stored in a buffer next to the register retirement map table and can be accessed in parallel to the physical register *checkpointed* bits. There is no ISA overhead associated with this technique.



(a) Normalised IPC

(b) Occupancy reduction expressed as a percentage of the maximum available

(c) Dynamic energy savings

(d) Static energy savings

Figure 6.6. Individual commit releasing schemes.

6.3.4 Instruction Tagging

The final commit releasing technique has the largest ISA impact and requires two spare bits in each instruction, one for each source register. If either bit is set, it indicates that this instruction is the last consumer of the register. The microarchitecture changes needed to implement this scheme are the same as for commit NOOPs in section 6.3.1. Additionally, the hardware must be able to check each source register's last consumer bit and release the corresponding physical register if set.

The compiler analysis for this approach involves constructing the control flow graph and liveness information as in section 6.3.1.2 but with one difference: each node in the control flow graph must be a separate instruction. Then, register R is tagged by the instruction in node N if the following condition holds:

Scheme	*Microarchitecture changes*
Commit NOOPs	At least 3 ports to the retirement map table and physical registers' *checkpointed* bits
Branches	At least 1 port to the retirement map table and physical registers' *checkpointed* bits
Procedure boundaries	Several ports to the retirement map table and physical registers' *checkpointed* bits, plus a buffer to record the registers to release
Tagging	Several ports to the retirement map table and physical registers' *checkpointed* bits

Table 6.1. Microarchitecture changes for commit releasing schemes.

$$R \in read[N] \wedge R \notin liveOut[N]$$

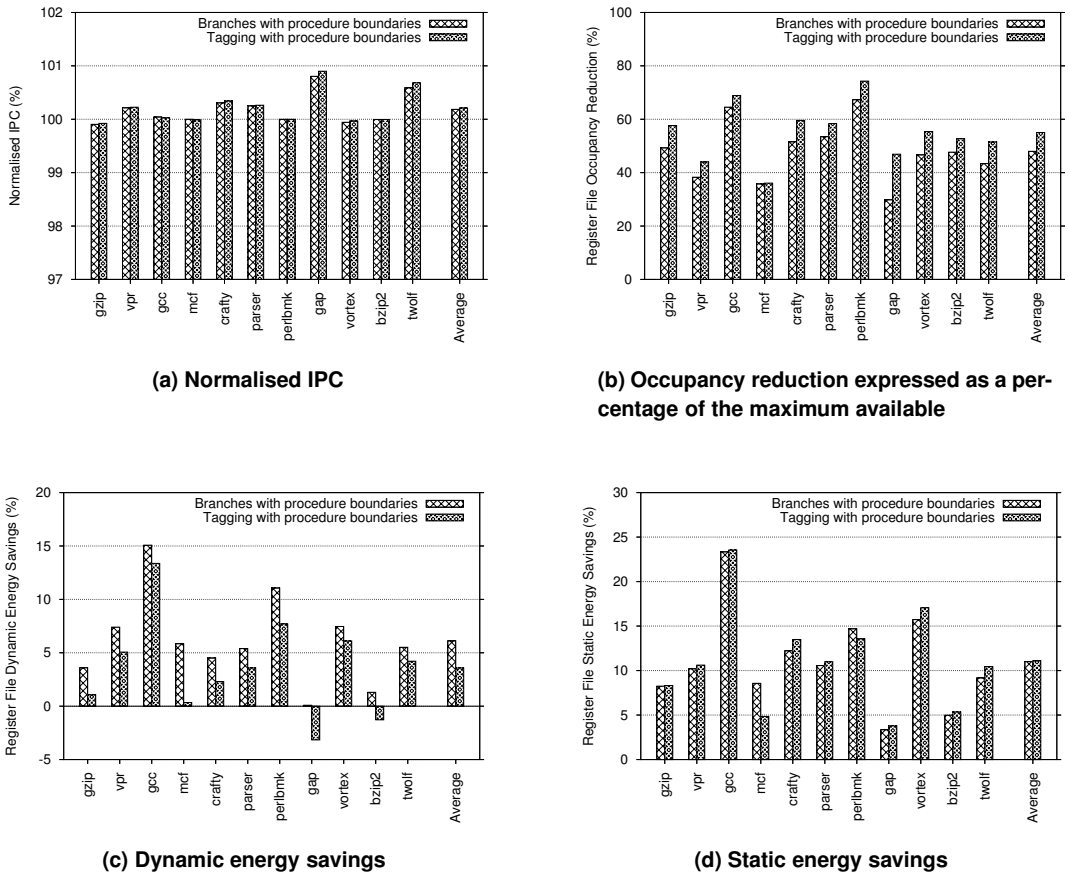
Register R must be a source register of the instruction in N and it must be its last use, i.e. it must not be live immediately after N.

6.3.5 Combined

By combining several commit releasing techniques, it my be possible to take advantage of different approaches. Procedure boundary releasing can be combined with all other schemes. However, it does not make sense to combine releasing on branches with tagging because they release the same registers: those that are tagged just release slightly earlier (they do not have to wait for the branch to commit). Hence, two new techniques are evaluated: branch and procedure boundary releasing; and tagging and procedure boundary releasing.

6.3.6 Results

The results from the commit releasing techniques are shown in figure 6.6. Figure 6.6(a) shows that most benchmarks experience a slight increase in performance. Figure 6.6(b) shows that tagging the last use of each register gives the most register file occupancy reduction, gaining 42% of the benefits on average. This translates to 3% dynamic and 9% static energy savings. Procedure boundary releasing, on the other hand, saves more energy (6% dynamic and 10% static) even though it has a lower occupancy reduction. This is because the tagging scheme checks the physical register *checkpointed* bits many more times than when only releasing at procedure boundaries, incurring a greater dynamic energy overhead.



(a) Normalised IPC

(b) Occupancy reduction expressed as a percentage of the maximum available

(c) Dynamic energy savings

(d) Static energy savings

Figure 6.7. Releasing at procedure boundaries combined with other commit releasing schemes.

Table 6.1 gives a summary of the microarchitecture changes required to support the four commit-based releasing techniques that have been proposed. Procedure boundary releasing, for example, requires enough ports to the register retirement map table and the physical registers' *checkpointed* bits to enable it to release a set of registers early. It also requires a buffer to record the set of registers being released early.

The results in figure 6.7(b) show that combining several commit-based early releasing techniques can further reduce the register file occupancy. In fact, releasing registers through tagging the last use combined with procedure boundary releasing gains 55% of the savings, almost the same amount as the hardware oracle when it releases at commit (it achieves 57%). This shows that using the compiler with simple microarchitecture and ISA changes can reduce the register file occupancy almost to the limit of that which is achievable. This translates into savings of

Scheme	*ISA changes*	*Benefit*	*Dynamic*	*Static*
Commit NOOPs	None	5%	4%	4%
Branches	Five free bits for one register in each branch instruction	30%	5%	8%
Procedure boundaries	None	37%	6%	10%
Tagging	A free bit for each source register in each instruction	42%	3%	9%
Branches and procedure boundaries	Five free bits for one register in each branch instruction	48%	6%	11%
Tagging and procedure boundaries	A free bit for each source register in each instruction	55%	4%	11%

Table 6.2. Summary of commit releasing schemes.

4% dynamic and 11% static energy.

Table 6.2 shows the ISA changes required from using each commit-based approach, benefits, in terms of occupancy reduction from baseline to hardware oracle and energy savings that result.

6.4 Issue Releasing

This section describes and evaluates issue-based early release techniques which release registers upon the issue of a consumer, once it has read the data. Releasing a register at the issue, rather than the commit, of its consumer potentially gives greater occupancy reduction at the cost of increased hardware resources.

Figure 6.8 shows the cumulative benefits that can be gained from issue-based releasing schemes. The x-axis shows the maximum number of uses a register has before it is released early. So, for example, when releasing registers with 4 uses or fewer at commit, 37% of the register file occupancy savings are obtained. However, releasing at the issue stage of the pipeline achieves a 77% reduction.

Issue-based releasing allows the benefits to be quickly realised. In fact, releasing all registers that are only used once (one-use) gains 57% of the occupancy savings. Releasing two-use as well gains 72%. This section only considers releasing one-use and up to two-use registers. Figure 6.8 shows that there are benefits to releasing three-use registers and beyond, but

experiments have shown that in practice these are hard to obtain.

The remainder of this section is structured as follows. Section 6.4.1 describes the microarchitecture changes and compiler analysis required to release one-use registers early. Section 6.4.2 then extends this to two-use registers. Section 6.4.3 considers the use of inter-procedural analysis to early release one-use register that are live across procedure boundaries. Finally, section 6.4.4 presents the results from these schemes.

6.4.1 One-use Registers

Previous work [23, 17] has discovered that many registers only have one consumer instruction. This can be exploited by releasing these registers (henceforth referred to as one-use registers) after the issue of their only consumer because they will not be read again along any control path without first being redefined. As figure 6.8 shows, approximately 57% of the benefit could be achieved from releasing such registers.

6.4.1.1 Compiler Analysis

The compiler analysis for identifying one-use registers is based on simple data-flow and liveness analysis. Liveness information is constructed in the same manner as commit releasing with commit NOOPs (see section 6.3.1.2). The only difference is that, in this case, each node in the control flow graph contains just one instruction.

To determine whether a register is one-use, the algorithm considers all its uses in the data dependence graph, their successor nodes in the control flow graph and the registers live into them. If the register under consideration is live into any successor of one of its uses then it cannot be a one-use register. This is because it will be used again after that use along some control path and hence is used more than once. If none of the successor nodes of its uses have the register in their live-in set then it is only used once.

Defining $uses[R]$ as the set of uses of register R in the data dependence graph means R is one-use if the following condition holds:

$$\forall U \in uses[R], \forall S \in succs[U] : R \notin liveIn[S]$$

The first task of the compiler's register renaming pass is to rename all registers to virtual registers to separate out different uses of the same logical register. This is because the first time a register is defined it might be used just once, but the second time it may be used many times

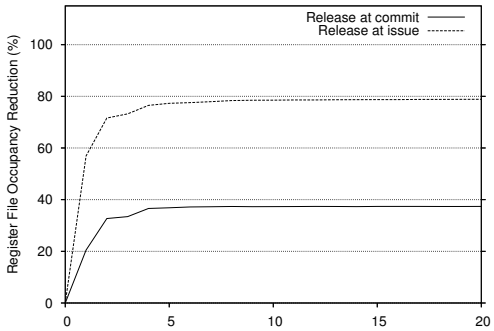


Figure 6.8. The cumulative increase in occupancy reduction from baseline to hardware oracle, achieved by releasing registers with a fixed number of consumers.

before a redefinition. Renaming to virtual registers assigns a unique virtual register number to each new definition, completely removing the link between different register definitions.

The next step is to recreate the interference graph which is then coloured with registers to get the final code. This graph colouring can never introduce spill code because the same number of logical registers are available for allocation as there were before they were renamed to virtual registers.

A standard graph colouring technique is used, as described by Appel [10] which creates a set of pairs of nodes (virtual registers) that share an edge in the interference graph. It also records the number of edges each node has; its degree.

The next stage is to determine the order in which nodes will be coloured: nodes with high degrees are first. A greedy algorithm is used to select the node with the highest degree and colouring proceeds from a fixed order of registers. One-use registers are allocated from one end of the ordering, multi-use from the other. Registers close to the end of the ordering are preferred so as to keep the total number used to a minimum.

These two ordering allocations can, of course, meet and overlap. In this case the register is considered a multi-use register to guarantee correctness. The maximum number of multi-use registers ever needed across the entire program is recorded. From this, the compiler can safely determine the number of one-use registers. Experiments show that the entire ordering is never needed for multi-use values and that there is always room for at least 5 one-use early-release registers. However, if the compiler wanted to change this, it could easily do this through the use of a special NOOP, called a usage NOOP which could encode the new number of one-use registers.

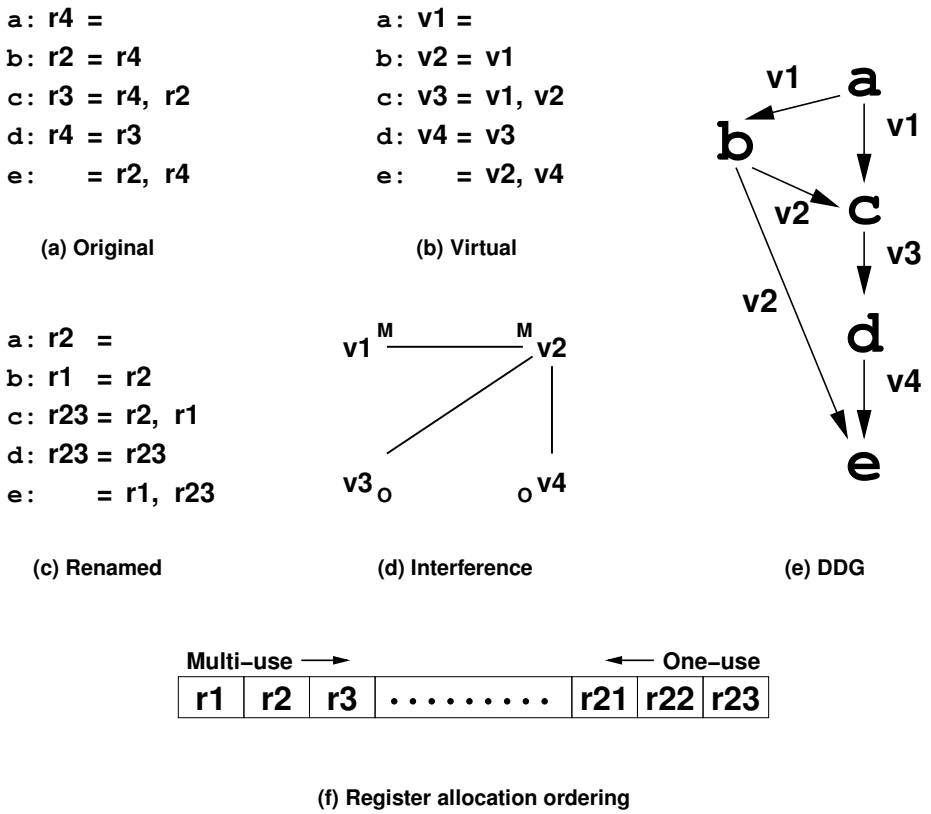


Figure 6.9. Overview of compiler pass.

6.4.1.2 Example

Figure 6.9 shows an example of the whole process of one-use register identification and register allocation. The original instructions are shown in figure 6.9(a) where r4 is multi-use the first time it is defined and one-use the second. Figure 6.9(b) shows the same code again after the registers have been renamed to virtual registers. The data dependence graph is constructed and is shown in figure 6.9(e) where the edges are annotated with the virtual register that is common between definer and user.

After one-use registers have been identified, the register interference graph is constructed as shown in figure 6.9(d) where the virtual registers are annotated with an *O* if they are one-use or *M* otherwise (i.e. they are multi-use). For example, v1 is a multi-use register as it is used in instruction b and c while v4 is only used in instruction e. As can be seen, register v2 interferes with all registers so has degree 3. There is no other interference so the others have degree 1. Figure 6.9(f) shows the ordering of logical registers that is used to allocate registers. One-use

registers are allocated from the right, whereas multi-use registers are allocated from the left.

Colouring proceeds with the highest degree first, so v2 is allocated a hard register from the multi-use end. It becomes r1. All other nodes have degree 1 so allocation occurs in lexical order. Virtual register v1 is multi-use and cannot be r1 because of the interference with v2, so it becomes r2. Virtual register v3 is one-use so gets a hard register from the opposite end of the ordering, r23. Likewise, v4 is one-use and so is also allocated from this end. In this case there is no interference with v3 so it can also be allocated to r23. The final code after allocation of all four registers can be seen in figure 6.9(c), where three logical registers are needed in both the original and renamed versions. Only 1 one-use register is needed here (r23) and the inserted usage NOOP containing the value 1 informs the hardware of this.

6.4.1.3 Microarchitecture Changes

Only simple changes to the microarchitecture are needed to support the early release of one-use registers in the issue stage of the processor. The register dispatch map table has a single bit added to each logical register entry which is called the *early_release* bit. This is used to determine, at dispatch, whether a register can be released early or not. These bits are set through the use of a usage NOOP which is stripped out of the instruction stream at dispatch.

The reorder buffer is augmented with two extra bits per source register to enable early releasing: an *early_release* bit and a *did_early_release* bit. When an instruction dispatches, the processor copies the *early_release* bit for each of its source registers from the dispatch map table into the *early_release* bits in the reorder buffer.

When the instruction issues, the *checkpointed* bit of each source physical register is read at the same time as the relevant *early_release* bit from the reorder buffer. This is in parallel to reading the data held in the main part of the register. For early releasing to take place safely the *early_release* bit should be set and the *checkpointed* bit unset (to show that there is no valid data in the shadow cells of the physical register). If early releasing takes place then, in the following cycle, the *did_early_release* bit is set for the source registers that are released early.

When an instruction commits, the *did_early_release* bit for each source register is copied to the register retirement map table for use in the event of an interrupt or exception. The previous version of its logical destination register is also released, as described in section 6.2.2.

On a branch mis-prediction, some instructions that released registers early may be squashed. By consulting the *did_early_release* bits of instructions being squashed, registers that were released early and checkpointed can be restored so that the correct user can read the right data.

The overhead of this scheme, in addition to those described in section 6.2.2, is two bits per source register in the reorder buffer, so 512 bits in all. There is no impact on the ISA.

6.4.1.4 Static and Dynamic One-use Registers

The most basic technique for releasing one-use registers early is to provide a certain number of fixed, one-use registers. Through analysis of the register requirements of the benchmark programs, five of the registers can be reserved for use as one-use registers. This is called a static number of one-use registers because these five are only ever used as one-use registers to be released early, never as multi-use registers.

A more complex but flexible scheme adapts the number of one-use registers to the changing requirements of the program. The number of these registers is fixed on a per-procedure basis through the use of a usage NOOP. The number encoded in them sets the number of registers from the predefined ordering that will be one-use until another usage NOOP alters them again. This approach is called a dynamic number of one-use registers because the number changes as the program executes.

6.4.2 Two-use Registers

With the ability to release one-use registers early, the natural next step is to increase this to allow two-use registers (those with two consuming instructions) to also be released early. The compiler analysis must change to identify these registers. However, the changes are simple and only require that along every path in the control flow graph from a defining instruction, the register defined is used exactly twice before being redefined. Both a static and dynamic number of two-use registers are considered. In the static approach, found by experimentation, four registers are used for one-use values and a single register for two-use. In the dynamic case the number of one and two-use registers is encoded in a usage NOOP as before.

6.4.2.1 Compiler Analysis

The compiler analysis to find two-use registers is much the same as that used in section 6.4.1.1 to find one-use registers. The difference is that along every path through the control flow graph from each definition of a register, the register must be used exactly twice. More formally, if $defs[R]$ is the set of all nodes defining of register R and $numUses[R][N]$ the number of uses of R immediately after node N along any control path from N, then the following condition must hold:

$$\forall D \in defs[R] : numUses[R][D] = 2$$

As before, each node in the control flow graph is a single instruction, therefore this condition guarantees that R is used exactly twice after each of its definitions.

Logical registers are renamed to virtual registers, one-use and two-use registers are identified and the interference graph created as in section 6.4.1.1. Colouring proceeds from the fixed ordering, as before, with two small changes. When a dynamic number of one and two-use registers are being allocated the one-use registers are allocated first from one end of the ordering, then the two-use, with the condition that no register that has been designated one-use anywhere in the procedure can be allocated to a two-use register. This is to ensure that a register is either one-use, two-use or multi-use for the whole procedure.

In the case of allocating a static number of one and two-use registers, specific registers are designated one-use or two-use and these are used, if there is no interference, when allocating a register of the respective type.

6.4.2.2 Microarchitecture Changes

The changes to the microarchitecture needed to support the early release of two-use registers in addition to one-use are simple. The register map tables and reorder buffer are altered in a similar way to that described in section 6.4.1.3 and some further additions are also made. The register dispatch map table, instead of having an *early_release* bit, is given two extra bits per entry to indicate whether a register is one-use, two-use or multi-use. It also gets an extra bit, called the *last_user* flag, which indicates whether an instruction is the last user of a physical register. It is set to one for a newly renamed one-use register, or zero otherwise.

When an instruction dispatches, the *early_release* bits in reorder buffer entry are set if the corresponding logical source register is one-use or two-use (as indicated in the dispatch map table) and the *last_user* flag is set as above. Once the instruction has dispatched, the *last_user* flag is reversed. This means that a two-use register will only be released early once, by its second consumer instruction.

Along with a *checkpointed* bit for each physical register, several bits are needed to count the number of consumer instructions that have to issue. This is called the *consumer_counter* and for releasing two-use registers it only needs to be one bit. When an instruction issues it can release the physical register early if the *checkpointed* bit is unset, the *consumer_counter* is

zero and the *early_release* bit in its reorder buffer entry is set. This means that only the second consumer of a two-use register can release a physical register early and only if it issues after the first consumer. This may lead to missed opportunities to release a register early but is important so that recovery from branch mis-predictions is not too complex. In fact, recovery from branch mis-predictions is the same as described in section 6.4.1.3 for one-use registers.

The overhead of this scheme is three bits per entry in the dispatch map table, two bits in the reorder buffer per source, as before, and a bit per physical register. Again, there is no ISA impact for this early releasing scheme.

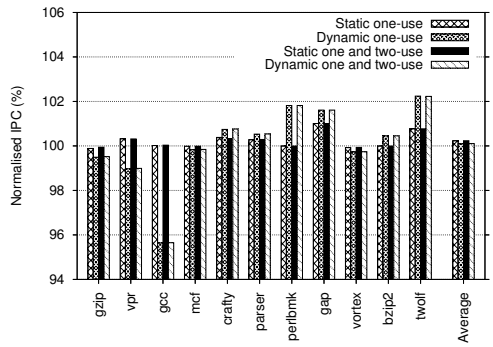
6.4.3 Inter-procedural Analysis

To allow the early release of one-use registers that are live across procedure boundaries, a simple form of inter-procedural analysis is performed. Registers used only once across procedure boundaries are identified on one pass then optimised in a second pass by renaming them to use one-use registers. Experiments have found registers that are used twice and are live across procedure boundaries are extremely rare, so these are not considered.

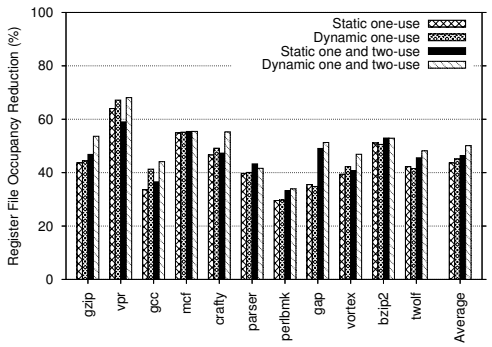
Special care has to be taken to allow dynamic procedure calls to take place. These are calls where the procedure target is only known at run-time and cannot be statically determined by the compiler. In the caller procedure, registers before a dynamic procedure call that are live across the boundary are not renamed because it cannot be determined whether they are one-use or not. Before each procedure that could be the target of a dynamic procedure call, extra `MOV` instructions are added to move the registers that are one-use into correct one-use registers for early releasing once they have been used. Static procedure calls do not execute these instructions; they are only used when a dynamic call targets the procedure. These extra `MOV` instructions are accounted for in all simulations.

6.4.4 Results

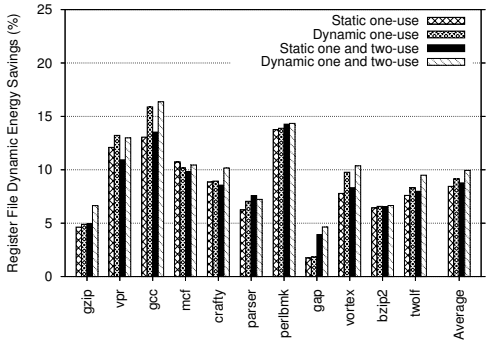
The results from releasing one and two-use registers are shown in figure 6.10. When there is a fixed number of one-use registers then the IPC rises slightly and some benchmarks experience a rise in IPC when the number of one-use registers is set dynamically. However, as figure 6.10(a) shows, certain benchmarks actually lose IPC in the dynamic one-use scheme due to the usage NOOPs, for example *gcc*. Considering one and two-use registers, again there is a slight performance improvement when the number of each is statically determined and fixed, but



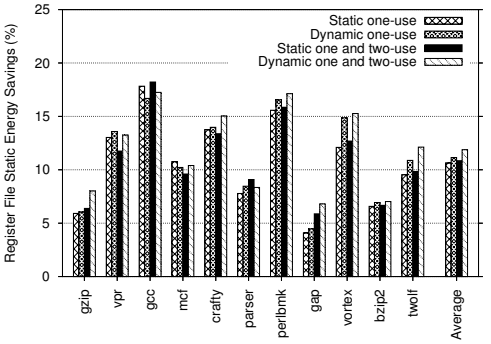
(a) Normalised IPC



(b) Occupancy reduction expressed as a percentage of the maximum available



(c) Dynamic energy savings



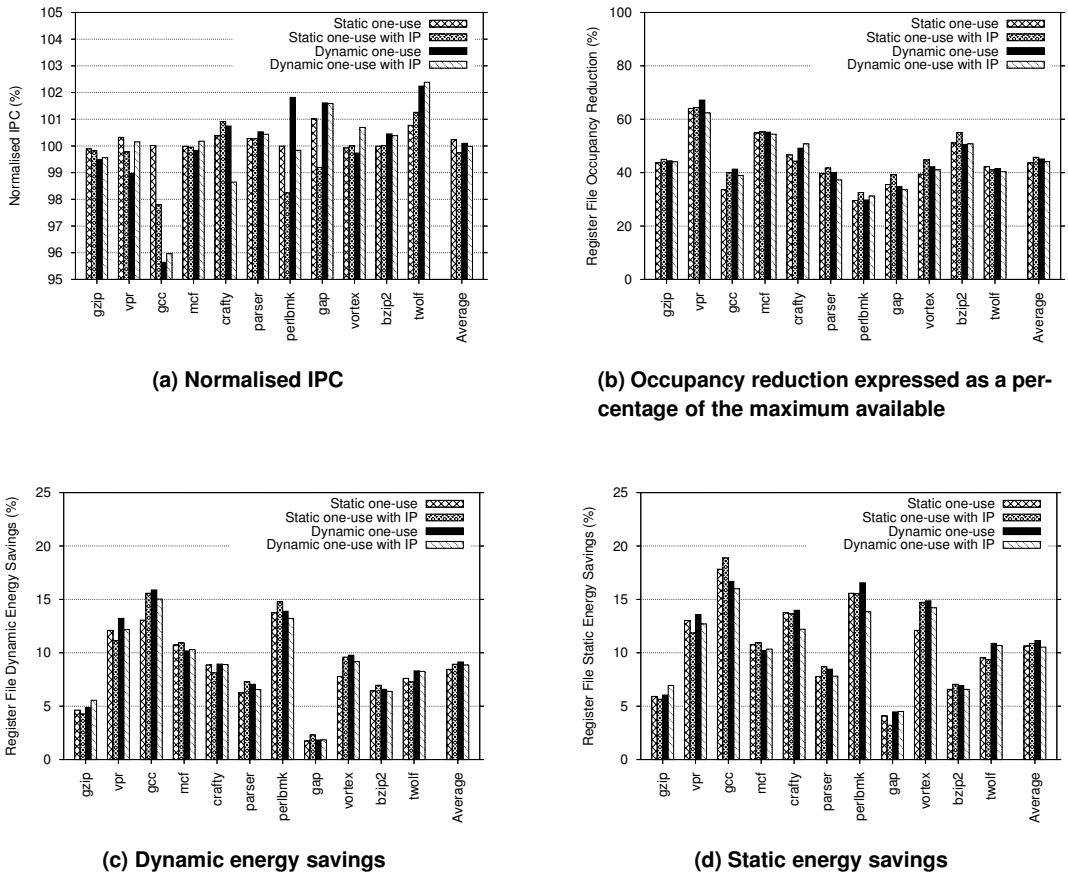
(d) Static energy savings

Figure 6.10. Releasing one-use and two-use registers.

when the usage NOOPs are used to dynamically set the numbers there is a slight IPC reduction in some benchmarks.

The changes in register file occupancy can be seen in figure 6.10(b) and it is clear that releasing one-use registers is beneficial. However, it appears that dynamically changing the number of one-use registers does not give much of an improvement over having a static number. This is because the majority of procedures only need a few one-use registers so providing only five in the static case achieves most of the benefits.

Releasing a static number of one and two-use registers brings some benefits. However, when a dynamic number are set, further improvement occurs. For example, *crafty* experiences an increase from 47% of the difference in occupancy in the static case to to 55% in the dynamic case. This is because some procedures have more than the one two-use register that is provided and this can be exploited when usage NOOPs are used to alter the number available.



(a) Normalised IPC

(b) Occupancy reduction expressed as a percentage of the maximum available

(c) Dynamic energy savings

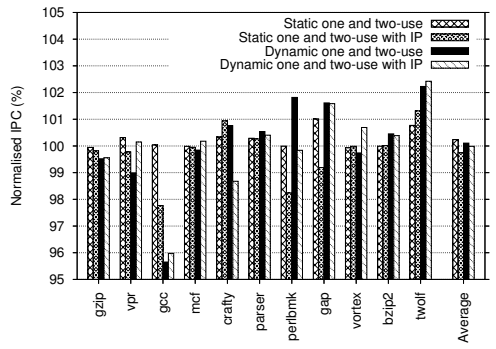
(d) Static energy savings

Figure 6.11. Releasing one-use registers with inter-procedural analysis.

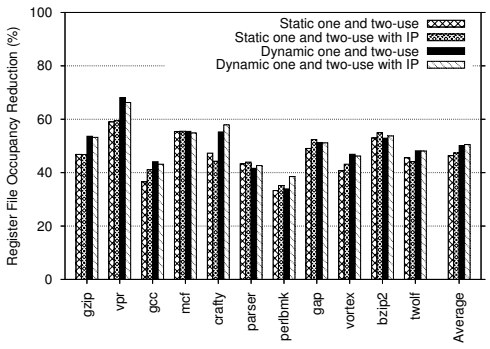
Dynamic energy savings are shown in figure 6.10(c) where it can be seen that reductions of over 8% are made by most schemes. Releasing a dynamic number of one and two-use registers brings dynamic energy savings of 10% and a static energy reduction of 12%, as shown in figure 6.10(d).

Not all the benefits that are available from releasing one-use registers, suggested by figure 6.8, are obtained. Some registers are one-use along one path through the control flow graph, but have several uses along another. They cannot be classified as one-use since it is not known at compile-time which path will be taken, so if the one-use path is actually the correct one when the program runs, an opportunity to release early is missed.

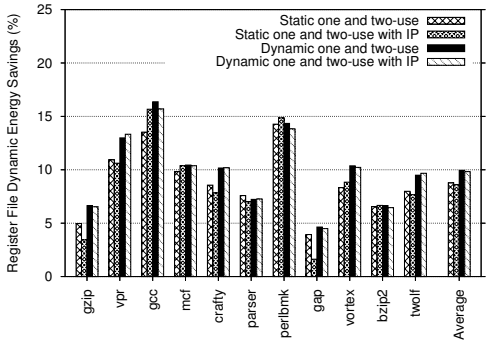
The effects of inter-procedural analysis on the issue-based early releasing schemes can be seen in figures 6.11 and 6.12. In general, there is a slight performance loss when using inter-procedural analysis due to the extra `MOV` instructions that must be executed on dynamic jumps.



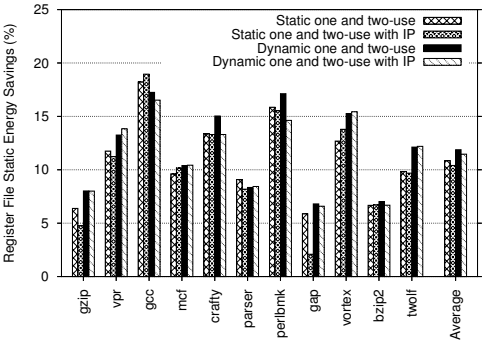
(a) Normalised IPC



(b) Occupancy reduction expressed as a percentage of the maximum available



(c) Dynamic energy savings



(d) Static energy savings

Figure 6.12. Releasing one and two-use registers with inter-procedural analysis.

This is shown in figure 6.11(a) for one-use registers and figure 6.12(a) for one and two-use registers. The slowdown is similar for both schemes.

The effects of inter-procedural analysis on register file occupancy are more complex. Figure 6.11(b) shows that the average occupancy reduction when using a static number of one-use registers actually increases when inter-procedural analysis is included. Yet, when there is a dynamic number, on average, the occupancy reduction falls. However, this type of analysis appears to affect each benchmark differently. For example, *twolf* experiences a smaller occupancy reduction when either a dynamic or static number of one-use registers are released early. The reasons for this changing behaviour are that some benchmarks have many dynamic jumps and thus execute many `MOV` instructions, increasing the register file occupancy. In addition to this, the code generated by Machine SUIF strictly obeys the caller and callee register conventions, almost eliminating the use of inter-procedural one-use registers.

Scheme	*Microarchitecture changes*	*Benefit*	*Dynamic*	*Static*
Static one-use	One bit per dispatch map table entry and four bits per reorder buffer entry	44%	8%	11%
Dynamic one-use	One bit per dispatch map table entry and four bits per reorder buffer entry	45%	9%	11%
Static two-use	Two bits per dispatch table entry, a bit per physical register and four bits per reorder buffer entry	46%	9%	11%
Dynamic two-use	Two bits per dispatch table entry, a bit per physical register and four bits per reorder buffer entry	50%	10%	12%

Table 6.3. Summary of issue releasing schemes.

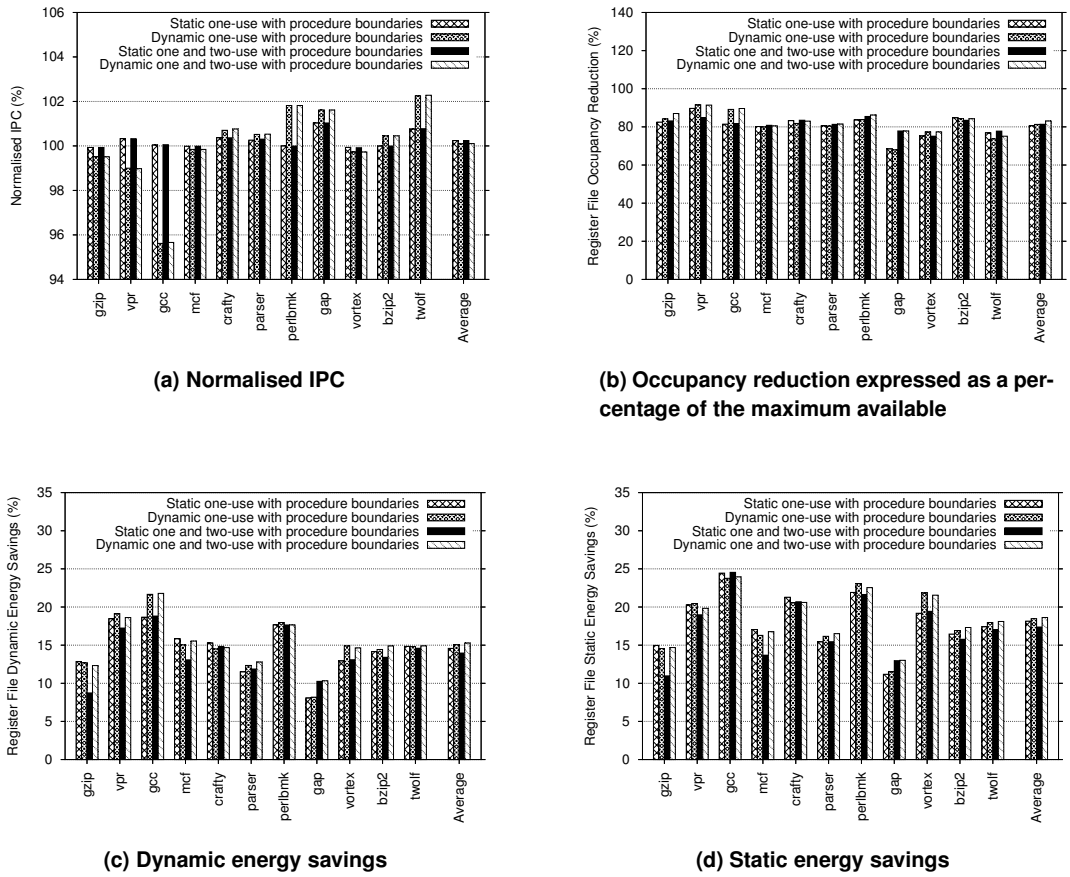
When considering one and two-use early releasing, the results of adding inter-procedural analysis appear to have similar effects on the register file occupancy, as shown in figure 6.12(b). Here there is an increase in the occupancy reduction when inter-procedural analysis is used, on average. However, in general the results are disappointing and do not bring significant increases in dynamic energy savings, as shown figures 6.11(c) and 6.12(c). Lower static energy savings are recorded by the schemes with inter-procedural analysis due to the increase in execution times.

A summary of the issue-based releasing schemes is given in table 6.3. All techniques gain at least 44% of the register file occupancy savings, with dynamic two-use releasing reaching 50%. All approaches also need two *early_release* and *did_early_release* bits in the reorder buffer to determine whether early releasing for each source register can happen or has occurred, along with an *early_release* bit in the register dispatch map table. In addition to this, a *last_user* bit and *consumer_counter* are needed for the two-use schemes. Dynamic energy savings range from 8% to 10% and static from 11% to 12%.

6.5 Combined Approaches

This section evaluates techniques that combine issue and commit releasing, aiming to further improve occupancy reduction. Four issue-based and three commit-based schemes are combined together. The combinations vary in terms of their performance and hardware/ISA impact.

The four issue-based approaches considered are: static one-use (section 6.4.1); dynamic one-use (section 6.4.1); static one and two-use (section 6.4.2); and dynamic one and two-use (section 6.4.2). These are the schemes from section 6.4 without any inter-procedural analysis,



(a) Normalised IPC

(b) Occupancy reduction expressed as a percentage of the maximum available

(c) Dynamic energy savings

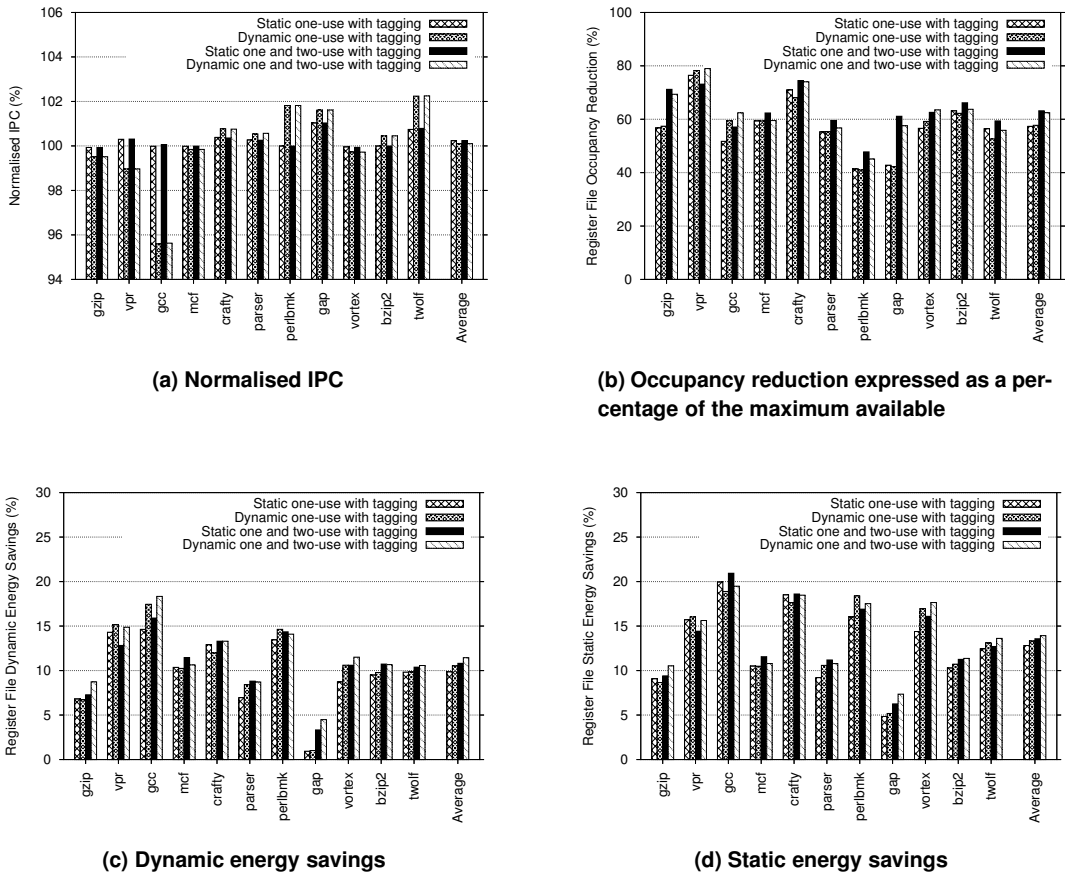
(d) Static energy savings

Figure 6.13. Combining issue releasing with procedure boundary commit releasing.

because they brought the most energy savings. The three commit releasing techniques are: procedure boundaries (section 6.3.3); tagging (section 6.3.4); and procedure boundaries with tagging (section 6.3.5). These also brought good energy savings and reductions in register pressure. The combined schemes work by releasing registers both at the issue and commit stages of the pipeline.

6.5.1 Results

Figure 6.13(b) shows the reduction in occupancy when commit-based procedure boundary releasing is added to each issue-based scheme. The results are within 17% of the limit of benefits when combining with dynamic two-use registers. Releasing static one-use registers and procedure boundary releasing achieves within 19% of the limit, or gains 81% of the available benefits, and this requires little modification to the microarchitecture.



(a) Normalised IPC

(b) Occupancy reduction expressed as a percentage of the maximum available

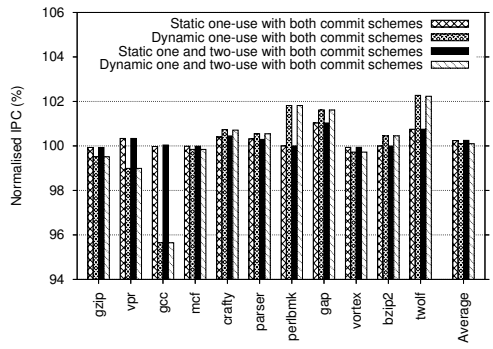
(c) Dynamic energy savings

(d) Static energy savings

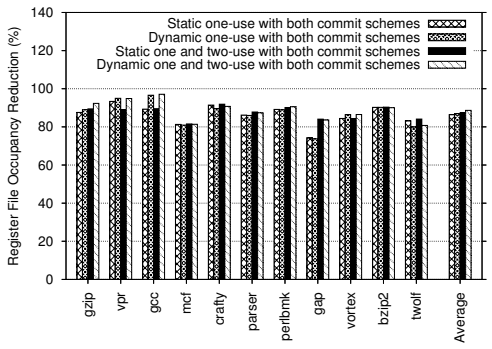
Figure 6.14. Combining issue releasing with tagging commit releasing.

Energy savings are increased using this scheme too. Dynamic energy reductions of 14% are achieved by most schemes (up from over 8%), as shown in figure 6.13(c). Increases in static energy savings (figure 6.13(d)) are also achieved, up from 11% to 17%.

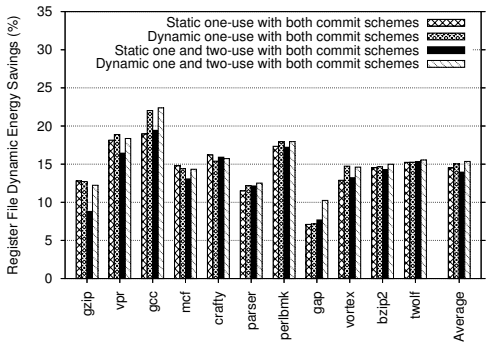
Combining commit-based tagging to the issue-based schemes achieves less impressive results, giving, on average, just around 60% of the available register file occupancy reduction, as shown in figure 6.14(b), with similarly less impressive dynamic energy savings (figure 6.14(b)). However, by using both commit releasing techniques with one-use and two-use releasing, savings close to 11% of the limit can be achieved, or 89% of the benefits gained. This is shown in figure 6.15(b). This requires major changes to the ISA and microarchitecture so should be used only if reducing the register file occupancy is a major concern. Otherwise, early releasing implemented with static one-use issue releasing and commit releasing on procedure boundaries gives a good trade-off between savings and changes to the microarchitecture.



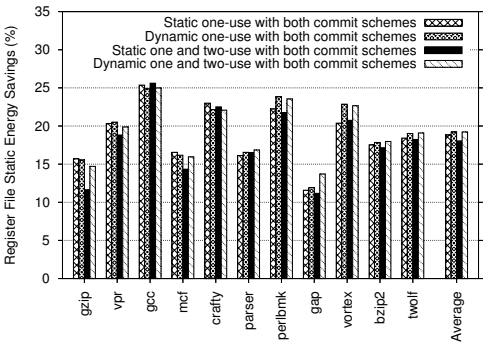
(a) Normalised IPC



(b) Occupancy reduction expressed as a percentage of the maximum available



(c) Dynamic energy savings



(d) Static energy savings

Figure 6.15. Combining issue releasing with tagging and procedure boundary commit releasing.

In general, tagging registers means that dynamic energy savings are limited due to the overhead in accessing the *checkpointed* bits frequently to determine whether a register can be released early. Although a smaller occupancy reduction is achieved using procedure boundary commit-based releasing, the infrequent checking of the *checkpointed* bits means that there is less of an overhead and consequently greater dynamic energy savings can be achieved.

Table 6.4 shows a summary of the recommended schemes, depending on limitations of the processor. If the ISA impact is to be kept to a minimum then using dynamic two-use issue-based releasing combined with commit releasing at procedure boundaries is suggested as there is no ISA impact. If the changes to the microarchitecture are to be minimal then only commit-based releasing should be used. The lowest occupancy is given by the dynamic two-use, issue-based releasing with both commit-based releasing techniques. An approach with

Requirement	*Scheme*	*Benefit*	*Dynamic*	*Static*
No ISA	Dynamic two-use with procedure boundaries	83%	15%	18%
Minimal hardware	Procedure boundary and tagging commit releasing	55%	4%	11%
No ISA and minimal hardware	Static one-use with procedure boundaries	81%	15%	19%
Lowest occupancy	Dynamic two-use with tagging and procedure boundaries	89%	15%	19%

Table 6.4. Summary of recommendations.

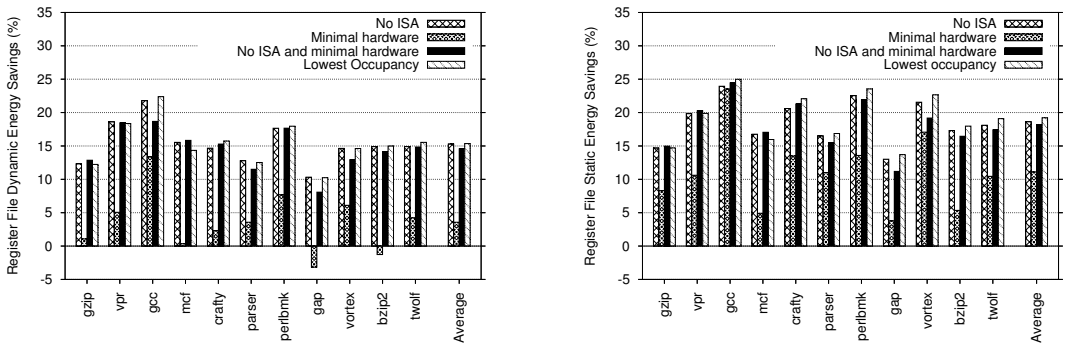
slightly less complex microarchitecture changes than the second scheme is using static one-use issue-based releasing, with commit releasing at procedure boundaries.

Figure 6.16 shows the normalised dynamic and static energy consumption of the recommended schemes. The *Minimal hardware* approach has the lowest energy savings with 4% dynamic and 11% static. This is because it only manages to gain 55% of the available benefits. The other schemes manage better. In terms of dynamic energy savings, shown in figure 6.16(a), both *No ISA* and *Lowest occupancy* obtain 15% savings, with the latter achieving just over 22% savings for *gcc*. Figure 6.16(b) shows the static energy savings resulting from these techniques. On average, at least 18% savings are made for the top three schemes. For individual benchmarks, *gcc* and *perlbmk* achieve the most savings, 25% in the best case.

6.6 Register File Size Sensitivity

To verify that the approaches are not specific to a particular register file size experiments are performed which decrease the number of registers from 112 (as in the original configuration) to 40 (when only 8 registers are available for holding non-committed values, the others being used to store the 32 logical registers). A comparison is made between two of the recommended techniques (*No ISA*: dynamic two-use with procedure boundaries; and *Lowest occupancy*: dynamic two-use with both commit schemes) and approaches proposed by Monreal *et al*. [62], Ergin *et al*. [27] and Martin *et al*. [58].

Figure 6.17(a) shows the IPC for the baseline (called *Baseline*), the techniques by Monreal *et al*. (*Monreal*), Ergin *et al*. (*Ergin*) and Martin *et al*. (*Martin*), the compiler-directed schemes (*No ISA* and *Lowest occupancy*) and the hardware oracle with early releasing at issue (*Hardware oracle issue*).



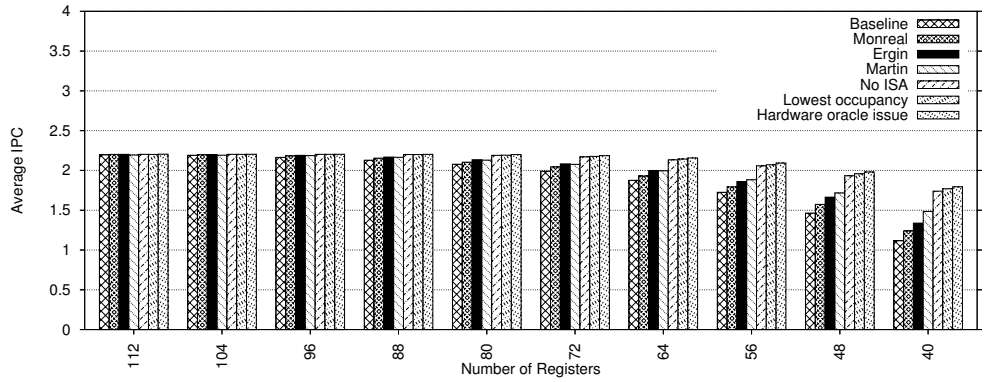
(a) Dynamic energy reduction **(b) Static energy reduction**

Figure 6.16. Normalised dynamic and static register file energy reduction for recommended compiler-directed early releasing schemes.

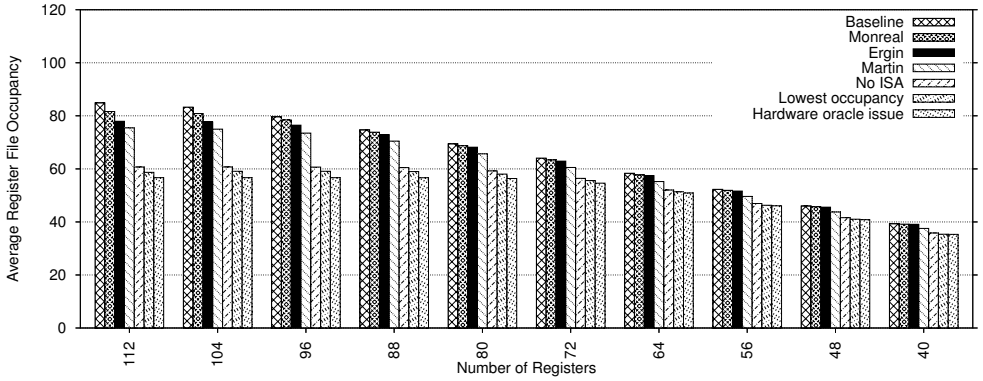
As can be seen from figure 6.17(a) and 6.17(b), both of the compiler-directed schemes considerably outperform the state-of-the-art hardware approaches across all register file sizes. For large register file sizes both schemes have better occupancy reduction whilst for small register files, they have superior IPC, approaching the oracle performance in both cases.

As the register file size decreases, so does the IPC, although the compiler schemes are always better than the others, almost achieving the IPC gains of the hardware oracle. In fact, with a register file size of only 88 entries, the IPC of *Lowest occupancy* is still better than the baseline with 112 entries. The *Monreal*, *Ergin* and *Martin* schemes do manage to increase the IPC of the baseline, especially for small register file sizes, but the compiler techniques presented in this chapter are consistently better. For example, when there are only 40 registers available then *Lowest occupancy* increases the IPC from 1.1 to 1.8, an increase of 61%, whereas *Martin* manages an increase of 32%, *Ergin* 20% and *Monreal* only 11%. When there are 64 registers, *Lowest occupancy* increases by 14% (from 1.9 to 2.1), *Martin* increases by 6%, *Ergin* by 6% and *Monreal* by 3%.

No energy results could be obtained for the *Monreal* technique so, instead, the average register file occupancy of each technique is compared as the register file size changes. The results are shown in figure 6.17(b). With large register file sizes the compiler schemes considerably reduce the register pressure, far more than *Monreal*, *Ergin* or *Martin*. In fact, even with a register file size of 80, *Lowest occupancy* reduces the register pressure by 17% (from 69 to 58) whereas *Martin* can only reduce it by 5% (to 66). Overall, the combined compiler-directed schemes are able to maintain higher IPC and reduce register pressure, allowing greater energy



(a) IPC



(b) Occupancy

Figure 6.17. Effects of reducing the register file size.

savings across all configurations.

6.7 Summary

This chapter has presented a detailed study of early register releasing with compiler support. It has proposed two types of compiler-directed early releasing, commit-based and issue-based, and shown that together they can reduce register pressure significantly.

Implementation of a hardware oracle shows that the best compiler analysis can achieve almost all the available benefits. The best compiler approach with significant but realistic changes to the hardware can make savings of 89% of the total. Alternatively, a scheme with no ISA impact and minimal hardware additions can achieve 81% of the maximum occupancy

reduction available. Comparing these techniques to two recently proposed hardware early releasing schemes and one previous compiler-directed approach shows that the register pressure can be significantly improved. Furthermore, these compiler schemes are superior across all register file sizes. In summary, compiler-directed schemes for early release can approach the limits of that which is possible, out-performing state-of-the-art hardware schemes whilst relying on less complex hardware.

Chapter 7

Combined Optimisations

This chapter presents combined schemes to reduce energy in the issue queue and integer register file through a mixture of issue queue throttling and early register releasing. It is structured as follows. Section 7.1 motivates the use of these two orthogonal techniques, section 7.2 describes the architectural modifications, whilst section 7.3 presents the results from four different limiting and early releasing combinations.

7.1 Motivation

Chapter 5 presents compiler-directed schemes to reduce the energy consumed in the issue queue whilst only suffering a small performance loss. This is achieved by limiting the number of instructions that can dispatch to the queue. This, in turn, reduces the occupancy of the banked issue queue, which is described in chapter 2. A beneficial side effect of this scheme is a reduction in register pressure leading to savings in the energy requirements of the banked register file.

Other schemes for register file energy reduction are presented in chapter 6. Here, registers are released earlier than usual once it is known that they will not be used again. This also leads to a reduction in register pressure and thus dynamic and static energy savings. If a lack of registers becomes a bottleneck to dispatch, these schemes can also increase performance, particularly for small register file sizes.

Using a combination of these schemes should bring issue queue energy savings, due to throttling the dispatch of instructions, and increased register file energy savings from a reduction in register pressure. The schemes are orthogonal because early releasing makes better use

of registers that are allocated whilst limiting the issue queue reduces the number allocated.

This chapter investigates combinations of issue queue throttling and early register releasing, evaluating each scheme in terms of performance, occupancy reduction and energy savings in the issue queue and register file.

7.2 Differences in Architectures

The only microarchitectural differences between the combined schemes presented in this chapter and their separate versions in chapters 5 and 6 concerns the operand fetch policies. The limiting schemes use dispatch-bound fetching whereas the early register releasing approaches use issue-bound fetching.

This chapter presents the combined schemes using both types of operand fetching, to allow comparisons with the separate approaches. With issue-bound operand fetching, a smaller issue queue is needed which has implications for the energy it consumes. In terms of dynamic energy, wakeups consume the largest proportion because there is no energy consumed in reading or writing operands into the queue, only their tags need to be added. For the static energy implications, there are fewer RAM cells meaning the banked part of the queue is smaller than with dispatch-bound fetching. Thus, the ratio between the banks and the selection logic increases and, since the selection logic is always on, the proportion of static energy that can be saved by turning banks off is reduced. This means that with issue-bound fetching, normalised dynamic energy savings will be higher than with dispatch-bound, but normalised static energy savings will be reduced.

Considering the register file, with dispatch-bound operand fetching registers are released at dispatch, if possible, or at issue otherwise, slightly reducing the register pressure over the issue-bound scheme. The structure of the register file does not change between schemes. Dispatch-bound operand fetching, therefore, reduces register pressure and allows for increased dynamic and static energy savings.

7.3 Results

In this section, four combined schemes are evaluated twice: once with operands fetched at dispatch and a second time with issue-bound fetching. In each case results are compared to the issue queue limiting and early register releasing schemes alone.

Scheme	*Limiting approach*	*Early releasing approach*
Minimal hardware	Block NOOPs	No ISA and minimal hardware
No ISA	Block NOOPs	No ISA
Lowest occupancy	Block tags improved	Lowest occupancy
No NOOPs	Block tags improved	No ISA and minimal hardware

Table 7.1. Summary of combined schemes.

Four combined schemes are shown in this chapter comprising a mixture of two fine-grained issue queue throttling approaches (*Block NOOPs* from section 5.5.2 and *Block tags improved* from section 5.5.3) and three early releasing techniques (*No ISA and minimal hardware*, *No ISA* and *Lowest occupancy*, summarised in section 6.5). A summary of the schemes can be seen in table 7.1.

The first approach (*Minimal hardware*) attempts to keep any hardware changes to a minimum as well as removing the need for ISA changes. The second (*No ISA*) considers the situation where changing the ISA alone is impractical. The third scheme (*Lowest occupancy*) combines the best approaches from chapters 5 and 6. The final scheme (*No NOOPs*) considers the effect of not using any special NOOPs, which avoids any side effects, such as increases in instruction cache misses.

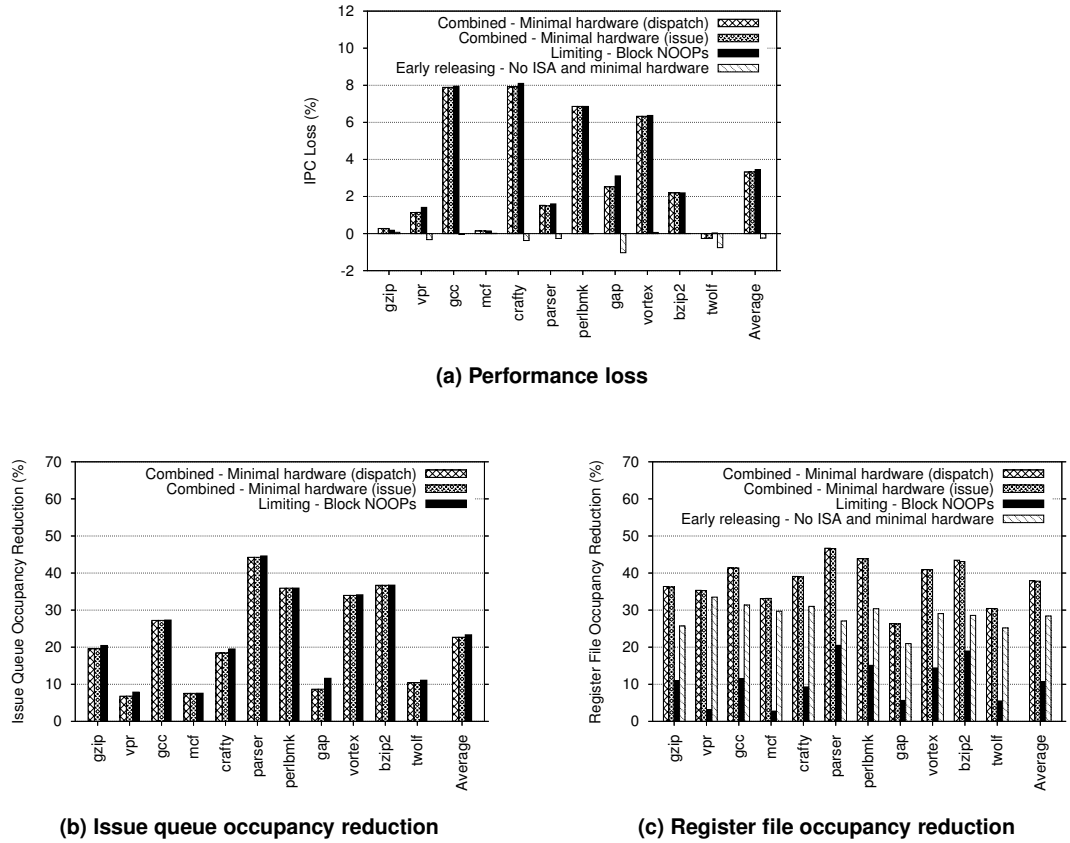
The graphs presenting the results of each scheme are normalised to the baseline architecture. Graphs showing the effects on the issue queue do not include techniques that only target the register file.

7.3.1 Minimal Hardware

This section evaluates the scheme called *Minimal hardware*, which requires minimal changes to the microarchitecture. This is because issue queue throttling is performed through the use of block NOOPs and early releasing is applied at procedure boundaries and through the use of a static number of one-use registers.

The performance loss of this scheme can be seen in figure 7.1(a). In general, the block NOOPs that limit the number of instructions entering the issue queue have a detrimental effect on performance which is hardly mitigated by the introduction of early register releasing. An exception is *twolf* which sees a negligible IPC increase of 0.3%. On average, early releasing helps performance but there is still a high IPC loss of 3.3%.

The issue queue occupancy reduction, shown in figure 7.1(b), is almost identical to that experienced when issue queue limiting alone is employed. Combining the schemes is beneficial



(a) Performance loss

(b) Issue queue occupancy reduction

(c) Register file occupancy reduction

Figure 7.1. Performance loss and occupancy reductions for *Minimal hardware*.

for the register file occupancy reduction (figure 7.1(c)) with average savings of 38% for both operand fetching combined approaches.

The issue queue dynamic (figure 7.2(a)) and static (figure 7.2(b)) energy savings are mostly unaffected by combining issue queue throttling and early register releasing. As discussed in section 7.2, reducing wakeups is more important with issue-bound operand fetching because they consume proportionally more of the issue queue's total energy. Hence, with issue-bound fetching, slightly more dynamic energy is saved compared with dispatch-bound operand fetching.

Large register file dynamic energy savings are achieved (shown in figure 7.2(c)), especially by *perlbmk* which has its dynamic energy reduced by 42% when dispatch-bound operand fetching is employed. In general, combining the two schemes has a beneficial effect on both dynamic and static register file energy, giving average dynamic savings of 25% and average

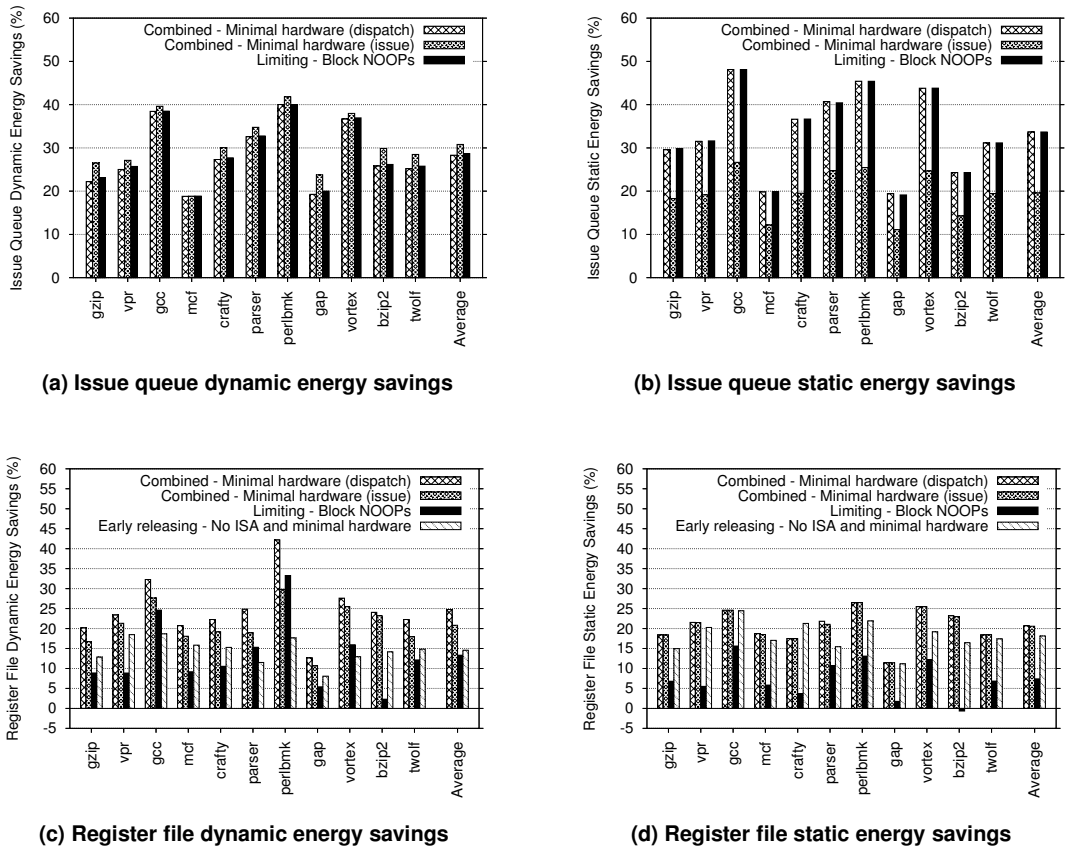


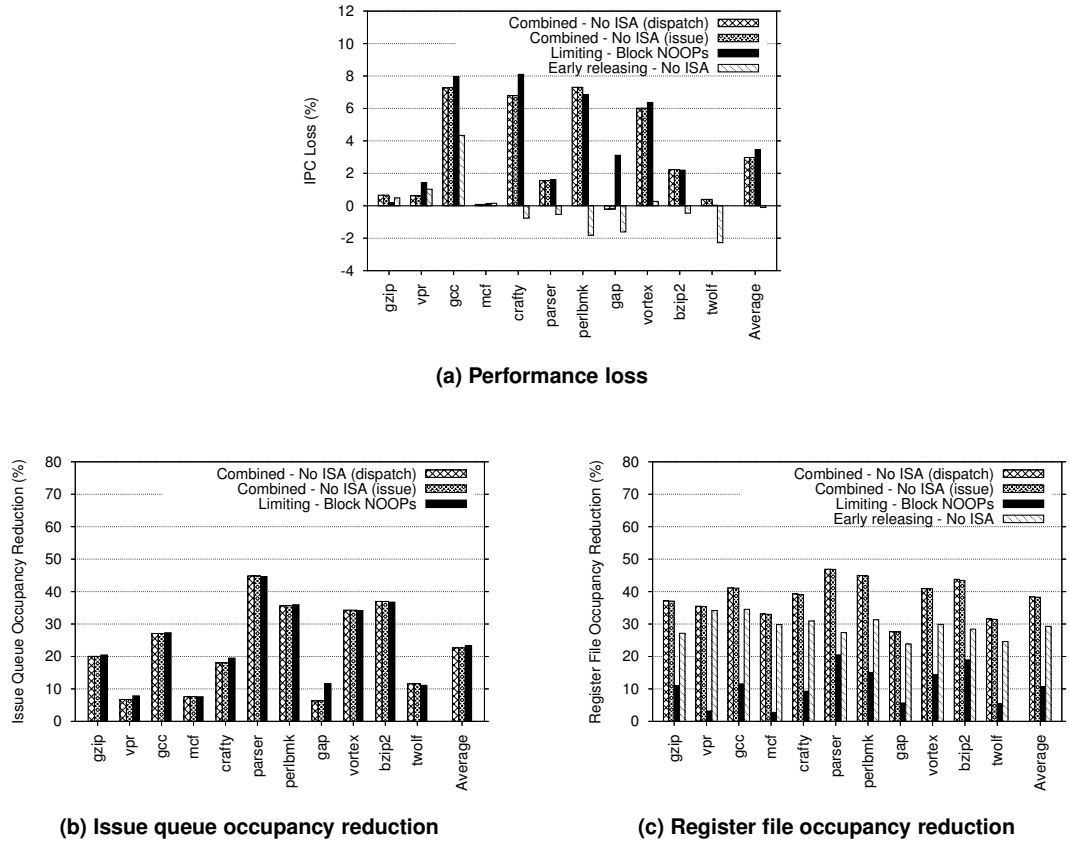
Figure 7.2. Energy savings for *Minimal hardware*.

static savings of 21% for dispatch-bound fetching.

7.3.2 No ISA

This section evaluates the *No ISA* scheme. Here, special NOOPs are used to limit the issue queue whilst registers are released early, either through the use of a dynamic number of one and two-use registers, or at procedure boundaries. This scheme has no ISA impact associated with tagging instructions, using instead two different special NOOP instructions: those to limit the issue queue (block NOOPs) and those to change the number of one and two-use registers (usage NOOPs). Although it could be possible to combine the NOOPs into a single instruction, this has not been evaluated.

The performance of this approach is shown in figure 7.3(a). The speedups achieved through early releasing help ameliorate the performance losses experienced by most benchmarks due



(a) Performance loss

(b) Issue queue occupancy reduction

(c) Register file occupancy reduction

Figure 7.3. Performance loss and occupancy reductions for *No ISA*.

to the issue queue limiting scheme, but some perform worse. For instance, *perlbmk* performs worse due to a 50% increase in instruction cache misses and branch mis-predictions compared with the limited issue queue version alone. However, at the other extreme, *gap* actually increases its IPC, largely due to a reduction in instruction cache misses over both the limited and baseline versions. The average IPC loss for this scheme is 3.0%.

Figure 7.3(b) shows that the change in issue queue occupancy reduction is negligible when adding early releasing to the issue queue limiting scheme. Figure 7.3(c) shows, as expected, that combining the early releasing and issue queue limiting schemes brings about increased register file occupancy reductions. In general, the early releasing and issue queue throttling schemes equally affect the issue queue and register file occupancy reductions.

In terms of energy savings, shown in figure 7.4(a), issue-bound operand fetching schemes again benefit the most from these combined approaches, although they save less static energy,

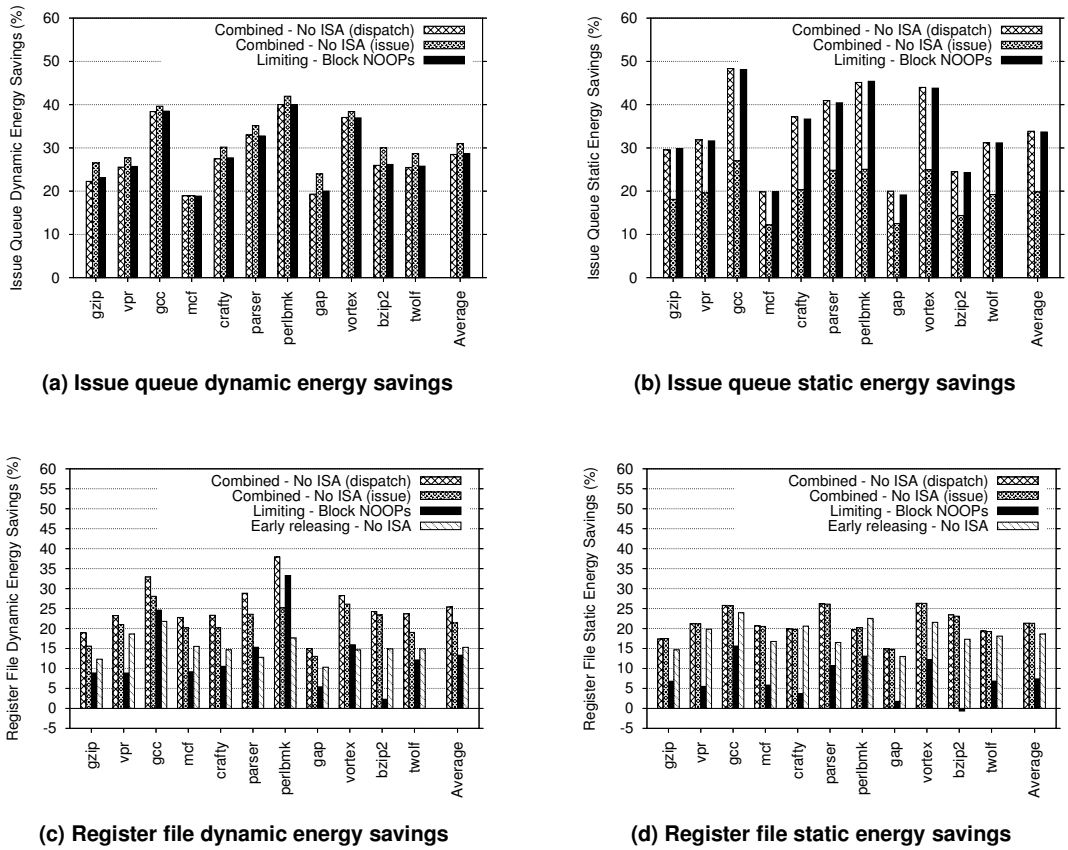


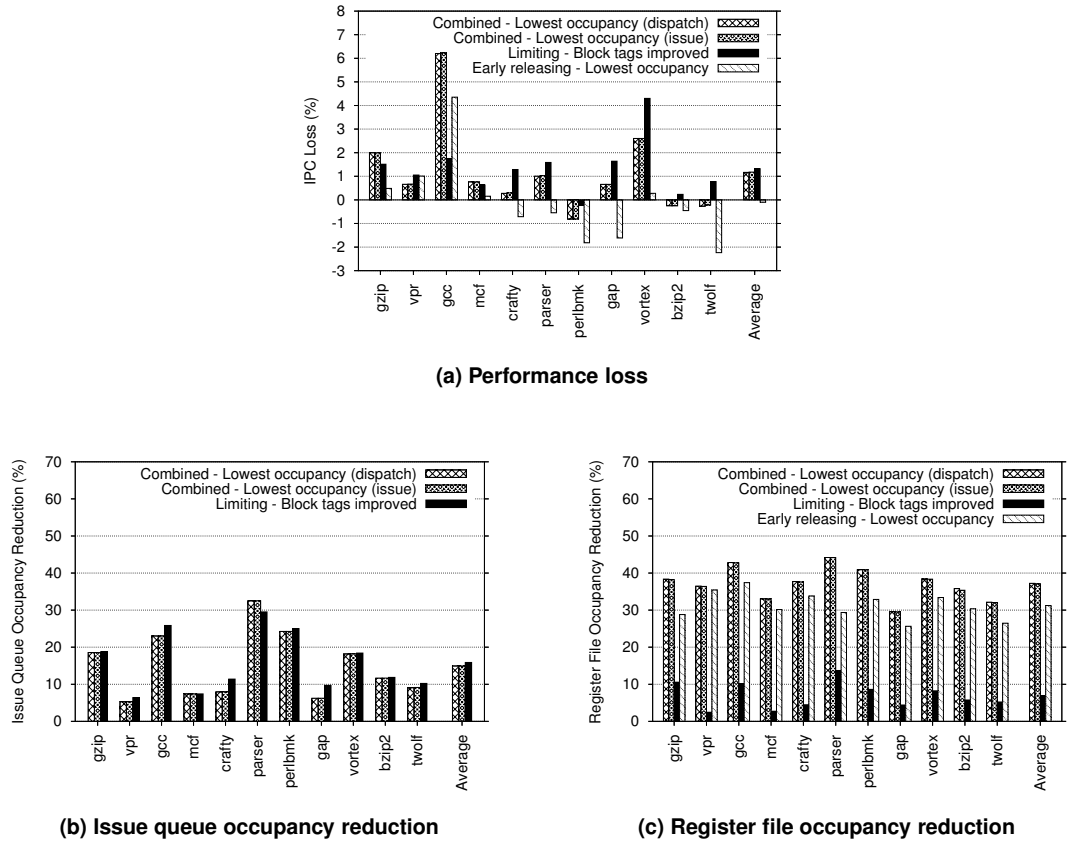
Figure 7.4. Energy savings for *No ISA*.

as seen in figure 7.4(b).

Finally, register file dynamic energy savings, shown in figure 7.4(c), shows that dispatch-bound operand fetching brings the most savings. Static energy savings (figure 7.4(d)) are larger for the combined approaches than in either the limiting or early releasing schemes alone, except for *perlbmk* and *crafty*. The former, because of its performance loss, has each of its banks on for longer meaning more static energy is consumed. The latter experiences a reduction in savings because, although it has more of an occupancy reduction than the early releasing scheme, it actually runs slower, again meaning banks are on for longer.

7.3.3 Lowest Occupancy

The *Lowest occupancy* scheme combines the *Block tags improved* issue queue limiting approach and the *Lowest occupancy* early register releasing technique. These are the schemes



(a) Performance loss

(b) Issue queue occupancy reduction

(c) Register file occupancy reduction

Figure 7.5. Performance loss and occupancy reductions for *Lowest occupancy*.

from chapters 5 and 6 that achieve the best occupancy reductions in the issue queue and register file with minimal performance loss. The *Block tags improved* scheme achieves high issue queue energy savings and *Lowest occupancy* creates high register file energy savings. The performance, issue queue and register file occupancies of the combined scheme are shown in figure 7.5.

Unfortunately, *gcc* suffers a large performance loss (6.2%) due to the fact that *Block tags improved* and *Lowest occupancy* run slower than the baseline on their own. Combined, there is even more of a slowdown due to the extra special NOOPs required to alter the number of one-use and two-use early releasing registers. This is also the case with *gzip* which suffers more of a performance loss when the combined schemes are used. Other benchmarks, for instance *vpr*, run slightly faster when both approaches are used. Often, the reduced register pressure from early releasing helps reduce the performance loss suffered when issue queue limiting is used

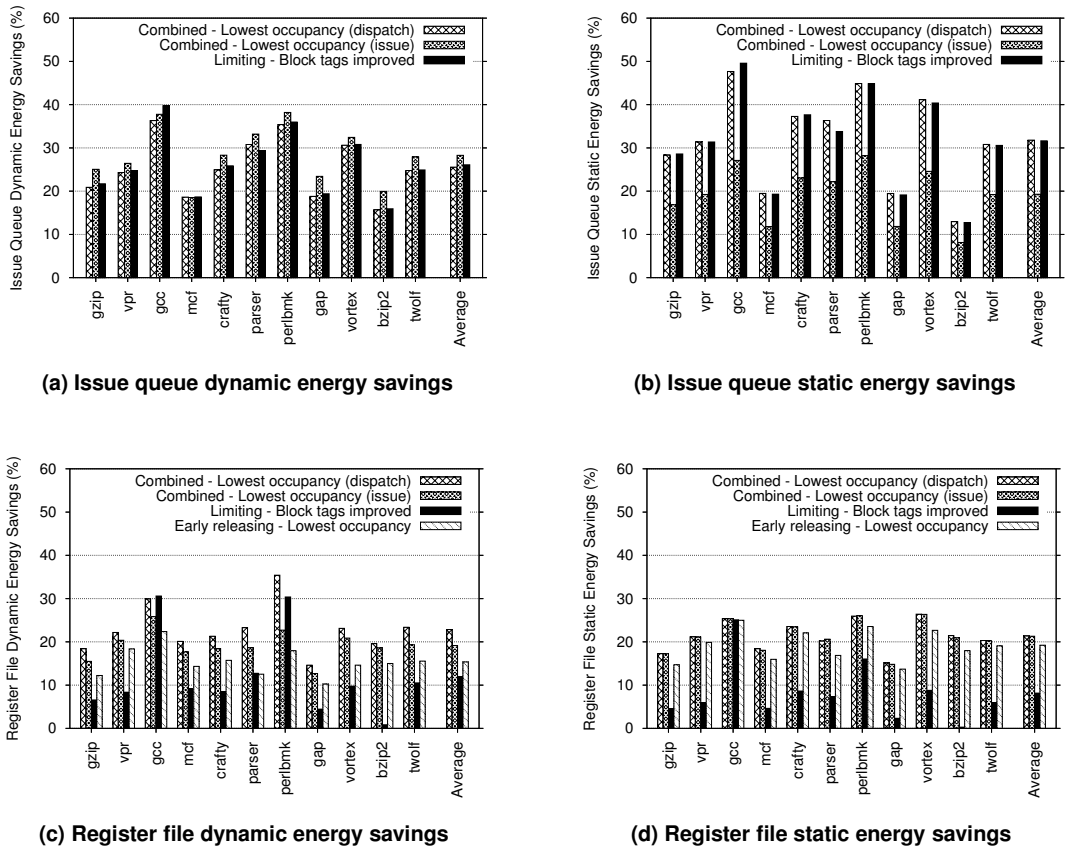


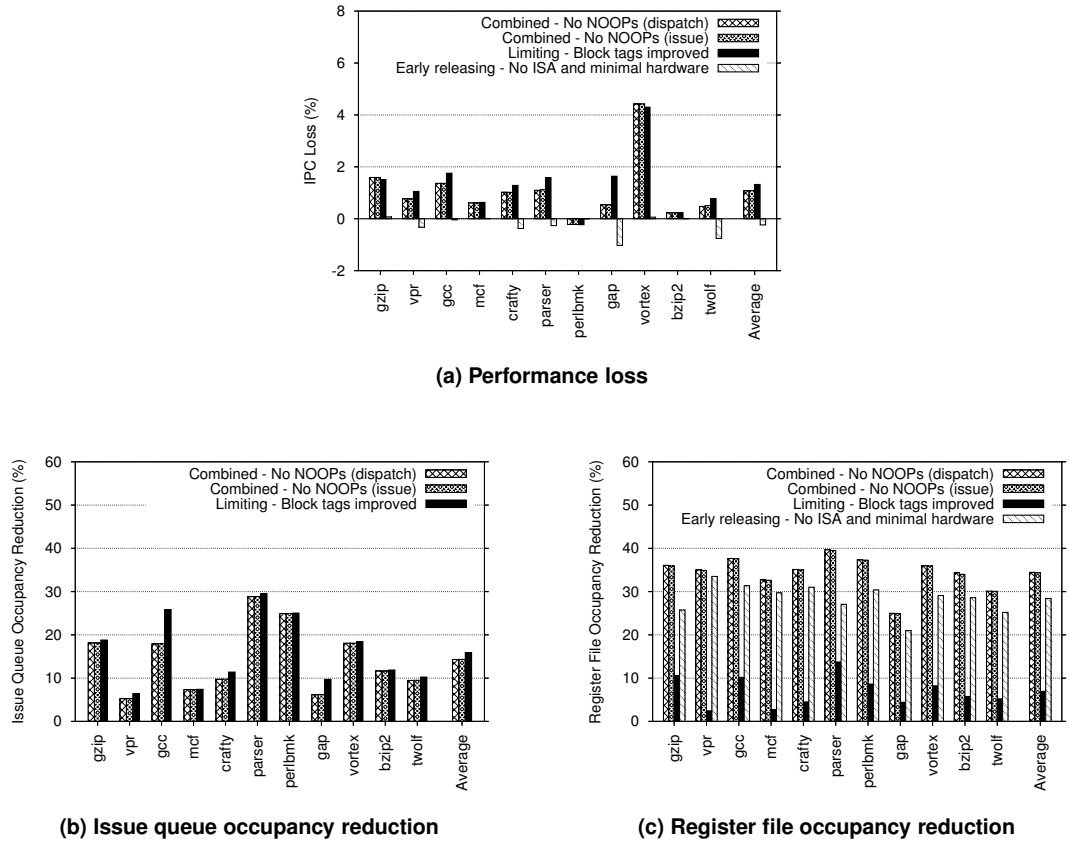
Figure 7.6. Energy savings for *Lowest occupancy*.

on its own. On average the performance loss is only 1.2%.

In terms of issue queue occupancy reduction, shown in figure 7.5(b), the results are varied. A larger occupancy reduction is experienced by *parser* when using the combined scheme compared with the limiting only approach, whereas all other benchmarks suffer a small loss. On average there is an issue queue occupancy reduction of 15% with the combined schemes compared to 16% with the issue queue limiting technique alone.

Reducing register pressure (figure 7.5(c)) allows more instructions to dispatch and increases the issue queue occupancy in some benchmarks because the instructions enter the queue too early and stay for several cycles. For others, it allows instructions to enter the queue and dispatch more quickly, actually reducing the issue queue occupancy.

The effect of occupancy on issue queue and register file energy is shown in figure 7.6. The benchmarks experiencing a greater reduction in their issue queue occupancy (e.g. *parser*) have



(a) Performance loss

(b) Issue queue occupancy reduction

(c) Register file occupancy reduction

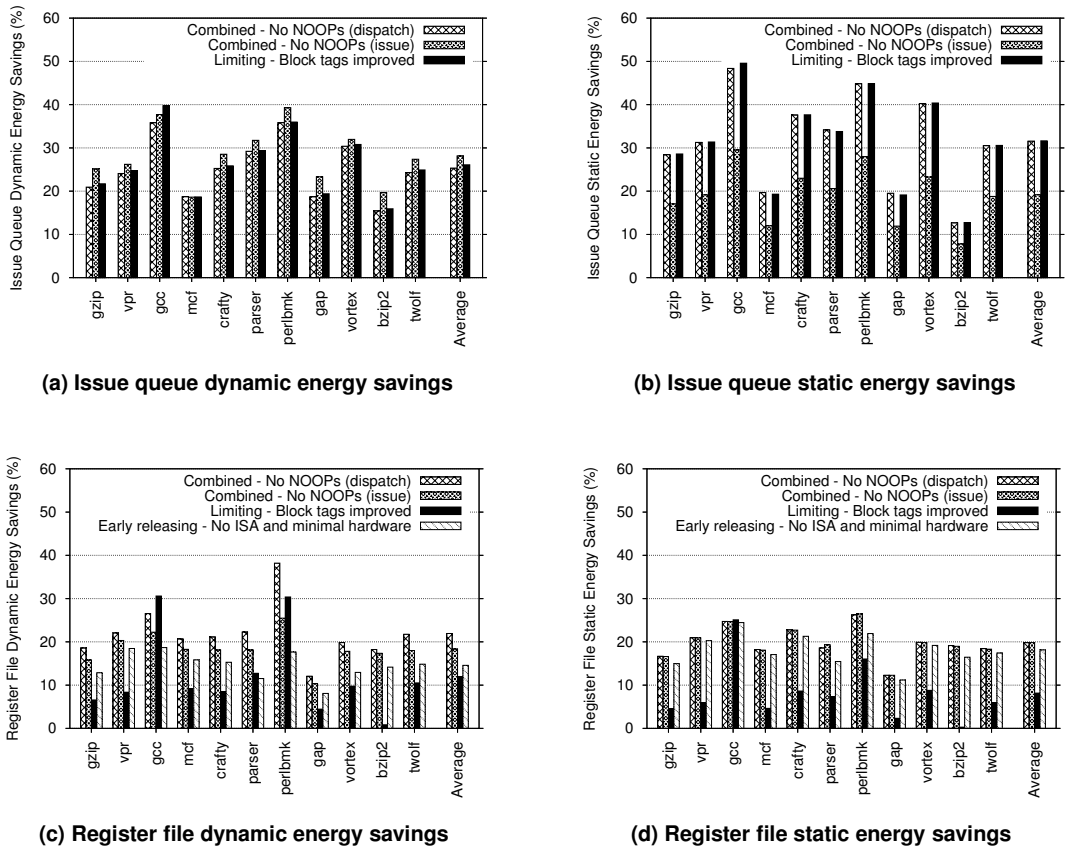
Figure 7.7. Performance loss and occupancy reductions for *No NOOPs*.

slightly higher dynamic energy savings (figure 7.6(a)). Those benchmarks experiencing less of a reduction in issue queue occupancy (for example *crafty*) save slightly less dynamic energy. However, the difference is only around 1%. Static energy savings in the issue queue remain approximately the same.

The register file energy savings for each benchmark is shown in figures 7.6(c) and 7.6(d). The greatest reduction in dynamic energy is achieved by *perlbmk* with 35% which also achieves large static energy savings of 26%, with an average savings. On average the benchmarks achieve large static energy savings of 21%.

7.3.4 No NOOPs

This final section evaluates the scheme called *No NOOPs*, which combines issue queue limiting through tagging with static one-use and procedure boundary early releasing. This scheme has



(a) Issue queue dynamic energy savings

(b) Issue queue static energy savings

(c) Register file dynamic energy savings

(d) Register file static energy savings

Figure 7.8. Energy savings for *No NOOPs*.

no side effects because no special NOOPs are added.

The performance loss for this approach is shown in figure 7.7(a). In most cases the IPC loss, incurred when using issue queue limiting only, is reduced by the combined schemes due to the decrease in register pressure from the early releasing. In some benchmarks, e.g. *gap*, the performance loss is significantly reduced, although other benchmarks, (*gzip* and *vortex*) experience a greater IPC loss. Overall the IPC loss is 1.1% for both operand fetching schemes.

The occupancy reduction in both the issue queue and register file are also shown in figure 7.7. There is a small change in issue queue occupancy reduction, with most benchmarks experiencing less of a reduction than when limiting alone is used. This is because more instructions along mis-speculated paths can be dispatched due to the decrease in register pressure.

The register file occupancy reduction is significantly better than when either issue queue limiting or early releasing is used in isolation. The best benchmark is *parser* with an occupancy

Scheme	*Operands*	*IPC Loss*	*IQ Dynamic*	*IQ Static*	*RF Dynamic*	*RF Static*
Minimal hardware	Dispatch	3.3%	28%	34%	25%	21%
	Issue	3.3%	31%	20%	21%	21%
No ISA	Dispatch	3.0%	28%	34%	25%	21%
	Issue	3.0%	31%	20%	21%	21%
Lowest occupancy	Dispatch	1.2%	26%	32%	23%	21%
	Issue	1.2%	28%	19%	19%	21%
No NOOPs	Dispatch	1.1%	25%	32%	22%	20%
	Issue	1.1%	28%	19%	18%	20%

Table 7.2. Summary of combined schemes.

reduction of almost 40%. On average it is just under 35% for both dispatch-bound and issue-bound operand fetching, compared to 7% when just limiting the issue queue or 28% when only using early register releasing.

The average issue queue occupancy reduction of 14% translates to 25% dynamic energy savings for the dispatch-bound, combined *No NOOPs* scheme, or 28% for the issue-bound version, as shown in figure 7.8(a).

The issue queue's static energy savings are shown in figure 7.8(b). There is a smaller saving with the issue-bound scheme (19% on average) because the selection logic, which is always on, consumes a greater proportion of the total issue queue static energy. There is no significant difference between the dispatch-bound version and the issue queue limiting approach.

Figures 7.8(c) and 7.8(d) show the energy savings in the register file using this combined approach. The greatest dynamic energy savings for all benchmarks are made when using a dispatch-bound operand fetch policy due to the ability to release registers occasionally as an instruction dispatches, as well as at issue and commit. This is especially apparent for *perlbmk*, which achieves 38% dynamic energy savings. Using an issue-bound policy, combined with limiting, is better than using early register releasing alone. Dynamic energy savings of 22%, for dispatch-bound, and 18% for issue-bound schemes are achieved.

The static energy savings, shown in figure 7.8(d), are similar. Here, the operand fetch policy makes little difference and greater energy savings are made than if early releasing alone had been employed. On average, for both combined schemes, 20% of the static energy is saved, compared to 18% when only using the early releasing scheme.

7.4 Summary

This chapter has presented the results of combining issue queue limiting with early register releasing. A summary of the schemes is presented in table 7.2. In general, the issue-bound operand fetching schemes save more dynamic energy in the issue queue, but less static energy. The dispatch-bound fetching schemes, however, save more dynamic register file energy. The performance loss for some schemes (*Lowest occupancy* and *No NOOPs*) is just over 1%.

In general, combining issue queue throttling and early register releasing is more beneficial to the register file than the issue queue. This is because the issue queue limiting schemes reduce the number of instructions dispatched

Chapter 8

Conclusions

This thesis has presented several compiler-directed schemes to reduce the energy requirements of superscalar processors without significantly impacting their performance. In particular, the occupancies of the issue queue and integer register file have been reduced by limiting the number of dispatching instructions and by releasing registers much earlier than usual. These schemes were then combined to give increased energy savings throughout the processor.

This chapter summarises the main contributions of this thesis in section 8.1. Section 8.2 performs a critical analysis of the proposed schemes, then, finally, section 8.3 suggests directions for future work.

8.1 Contributions

This thesis has targeted the issue queue and register file, the two most energy-consuming structures within a superscalar processor. Using simple compiler analysis, information is passed to the processor enabling it to determine the future requirements of each structure and save energy with negligible performance losses.

8.1.1 Issue Queue

The issue queue in a superscalar processor consumes a large amount of energy, mainly due to the waking of instructions' source operands every cycle by results returning from the functional units. Previous methods of reducing the energy consumption have proposed limiting the number of instructions present in the queue [5, 21, 32] and preventing wakeups to empty and ready entries [32].

This thesis has proposed a compiler-directed scheme to dynamically limit the number of instructions in the issue queue based on critical path analysis. The compiler knows which parts of the program are to be executed in the near future and can resize the queue without slowing down the critical path. This contrasts with hardware schemes which use past knowledge about a program to determine future requirements over a fixed sampling window. The compiler approaches proposed in this thesis use exact knowledge of the future and can adjust the queue size accordingly, meaning that the analysis is more precise and rapid phase changes can be accommodated easily.

Two compiler directed schemes have been evaluated in this thesis, both using either special NOOPs or tags on instructions to convey the resizing information. The compiler analysis considers two types of program structure: directed acyclic graphs (DAGs) and loops, where the analysis is altered slightly for each due to the parallel nature of loop execution within the processor.

The first compiler scheme inserts resizing information at the start of each DAG or loop informing the processor of the issue queue requirements for the whole structure. The second scheme places information at the start of every loop and basic block. The latter scheme results in the least performance loss (just 1.3% on average) with 15% reduction in issue queue occupancy. This translates into savings of 26% dynamic and 32% static energy, out-performing a state-of-the-art hardware scheme [5].

A side-effect of limiting the number of instructions dispatching to the issue queue is a reduction in register pressure. This increases opportunities to turn off empty register file banks, bringing dynamic and static energy savings to this structure as well.

8.1.2 Register File

Within the register file of a superscalar processor, physical registers are allocated as instructions are dispatched, written to when their producing instruction has executed, read by each of their consumer instructions, then released at the commit of the instruction redefining the logical register they represent. This time frame can include many cycles where the register is idle between the issue of the last consumer instruction and the commit of the redefining instruction.

This thesis has proposed releasing registers much earlier than usual, using the compiler to convey information about when they become idle. The schemes proposed release registers much earlier than other hardware releasing approaches which currently must wait for the redefining instruction to enter the pipeline to be sure that all consumer instructions have been

seen [62, 27]. The proposed techniques also release earlier than other software-based schemes which wait until the commit of the releasing instruction to perform the deallocation [58]. All approaches proposed in this thesis can reconstruct the precise processor state in the event of an interrupt or exception, something lacking in some previously proposed schemes.

A hardware oracle has been implemented to determine the extent to which early releasing can help reduce register pressure. An idealistic software-based approach compared to this oracle has shown that 94% of the benefits can be achieved through compiler-directed schemes releasing registers at the issue of the last consumer.

This thesis has explored the design space of compiler-directed early register releasing, based on simple data-flow and liveness analysis. It has considered commit-based schemes, which release registers upon the commit of a certain instruction, and those that are issue-based, which release registers when a consumer issues. Several proposed schemes have been compared to three state-of-the-art approaches and have been found to reduce register pressure by a greater extent when using a large register file, or produce greater performance increases when using a small number of registers. For the scheme with the greatest reduction in register pressure, 89% of the hardware oracle benefits can be achieved, translating into savings of 15% dynamic and 19% static energy.

Finally, early register releasing has been combined with issue queue limiting in several different ways to produce significant energy savings in the issue queue and register file with negligible performance losses.

8.2 Critical Analysis

One of the largest problems in passing information from the compiler to the microarchitecture is in determining the form it should take. Changes to the instruction set architecture (ISA) prevent backward compatibility and, in existing processors where there are few instructions with any free bits, difficult to incorporate effectively.

This thesis has proposed two methods of passing issue queue resizing information to the processor: special NOOPs and instruction tagging, and four methods of passing early register releasing information: implicit instructions, such as calls and returns; instruction tagging; special register numbers for one and two-use registers; and special NOOPs. Of these, using information implicit in instructions or using special register numbers are preferred as they do not change the ISA or have any side effects associated with them. Special NOOPs are also useful because they can contain a great deal of information and have no ISA impact, although

Scheme	*Issue Queue*		*Register File*		*Total*		*IPC Loss*
	Dynamic	*Static*	*Dynamic*	*Static*	*Dynamic*	*Static*	
Limiting - Abella	27%	30%	15%	11%	19%	18%	2.3%
Limiting - Block Tags Improved	26%	32%	12%	8%	17%	18%	1.3%
Early releasing - Ergin			9%	9%	2%	2%	0%
Early releasing - Martin			9%	13%	2%	3%	0%
Early Releasing - No ISA			15%	18%	3%	4%	0%
Early Releasing - Minimal hardware			4%	11%	1%	2%	0%
Early Releasing - No ISA and minimal hardware			15%	19%	3%	4%	0%
Early Releasing - Lowest occupancy			15%	19%	3%	4%	0%
Combined - Minimal hardware	28%	34%	25%	21%	24%	25%	3.3%
Combined - No ISA	28%	34%	25%	21%	24%	25%	3.0%
Combined - Lowest occupancy	26%	32%	23%	21%	22%	24%	1.2%
Combined - No NOOPs	25%	32%	22%	20%	21%	23%	1.1%

Table 8.1. Summary of energy saving schemes.

they can alter the instruction cache's behaviour and increase the number of instructions that are fetched into the processor. The most difficult method to implement is instruction tagging, although with future generations of processors, where more compiler-generated information is passed down to the hardware, these may become more realistic.

Limited inter-procedural analysis has been used in this thesis when considering the release of one-use registers across procedure boundaries, with negligible impact. The use of this type of analysis to aid the issue queue limiting schemes has not been implemented in the compiler. However, minor, hand-coded inter-procedural analysis applied to two benchmarks showed that there was potential for it to reduce the performance loss associated with these schemes where it is difficult to determine data dependences when control passes between procedures. This is difficult to implement in the current compiler framework but could be considered in a post-

linking optimisation pass.

Measuring the energy consumed in a processor is difficult and current models are not yet mature. Wattch [15] and Cacti [73], from which the energy measurements used by Abella and González [5, 4] were calculated, are standard for research into superscalar energy reduction. With integration into SimpleScalar [16], a widely used simulator in computer architecture research, the results produced are accepted as valid. This thesis presents normalised energy results to avoid problems assigning definite values to the energy consumed. The occupancy reductions achieved in the structures analysed are also presented, because they give a good indication of the potential energy saved through turning off empty entries.

Table 8.1 shows a summary of the results achieved by schemes proposed in this thesis along with several state-of-the-art techniques compared against in chapters 5 and 6. The dynamic and static energy savings are shown for the issue queue, register file and overall in the processor. The column on the far right shows the performance loss suffered by the schemes. The early releasing schemes do not target the issue queue and hence no results for energy savings are shown.

Considering the issue queue limiting schemes, *Abella* saves slightly more dynamic energy than *Block tags improved*, but also loses 2.3% performance, whereas *Block tags improved* only loses 1.3%. When looking at the early releasing approaches, similar amounts of energy are saved by the state-of-the-art schemes and those proposed in this thesis, although *Martin* does not implement precise interrupts and exceptions and the complexity of the extra hardware to perform early releasing with *Ergin* is not known. Overall, table 8.1 shows that combining issue queue throttling with early register releasing saves considerable energy within the processor. Each scheme saves at least 21% dynamic and 23% static energy, with performance losses ranging between 1.1% and 3.3%.

Finally, the work presented in this thesis has only targeted single-core superscalar processors. Trends towards multi-threaded, multi-core processors will present new challenges to architects of the future. However, the issue queue and register file will always be critical resources with high power dissipation. Schemes presented in this thesis can be adapted to different processor architectures with little difficulty due to the simple, generic nature of the compiler analysis which is not focused on one particular microarchitecture.

8.3 Future Work

This thesis has explored the design space of compiler-directed early register releasing schemes, proposing an approach that can gain 89% of the available reduction in register pressure. There is little further benefit to be gained from other early releasing schemes, but the register file itself could be the target for other compiler analysis.

Register caching schemes, described in chapter 3, attempt to limit the performance loss and complexity associated with a multi-cycle register file by providing a small, fast register cache to hold the values that are expected to be used most often in the near future. Hardware schemes must rely on the past to predict future usage patterns and move values speculatively into, or evict from, the cache based on various metrics. The compiler could help by passing exact information about the future register requirements to the processor, in much the same way as is proposed for early register releasing, and eliminate the need for speculation. This could bring energy savings in the backing store and performance improvements with a small register cache.

Register packing is another register file optimisation scheme that the compiler could help with. By indicating short values (those that do not use the whole bit-width provided) to the processor, complexity could be saved in the packing implementation.

Considering the issue queue, the compiler may be able to aid wakeup-free designs by communicating the number of consumers of a value or their distances from the producer. This would help the tracking of dependence chains through the pipeline, allowing for more efficient use of critical, issue stage resources.

Other areas of the processor could be targeted by the compiler using analysis of the data dependences and control flow information to save energy, for example the load/store queue and data cache which could benefit from analysis of memory instructions and data.

Finally, compiler analysis could be applied to different processor architectures to save energy, especially in the embedded domain where simpler hardware is used and the compiler is more critical. More complex processors, such as parallel and multi-threaded architectures, can also be considered which will each have different challenges in how the compiler can be applied to help these systems save energy and reduce power dissipation.

Appendix A

Energy Breakdown

Component	*Abbreviation*	*Energy (pJ)*
Decode (4 to 16)	4to16	3.8
Decode (3 to 8)	3to8	1.5 per bank
Register file address routing	rf_addr	1.3
Register file RAM wordline	rf_wl	0.5 per bank
Register file RAM bitline	rf_bl	4.6 per bank
Register file data to/from bank	rf_data	10.8 per bank
Register file output driver	rf_out	27.6
Issue queue address routing	iq_addr	0.9
Opcode RAM wordline	opc_wl	0.2 per bank
Opcode RAM bitline	opc_bl	1.5 per bank
Opcode data to/from bank	opc_data	3.6 per bank
Opcode output driver	opc_out	6.2
Operand RAM wordline	opnd_wl	0.5 per bank
Operand RAM bitline	opnd_bl	4.8 per bank
Operand data to/from bank	opnd_data	11.2 per bank
Operand output driver	opnd_out	27.6
Destination tag RAM wordline	dtag_wl	0.1 per bank
Destination tag RAM bitline	dtag_bl	0.4 per bank
Destination tag data to/from bank	dtag_data	0.9 per bank
Destination tag output driver	dtag_out	1.6
Source tag CAM wordline	stag_wl	0.1 per bank
Source tag CAM bitline	stag_bl	0.4 per bank
Source tag data to bank	stag_data	0.9 per bank
Source tag CAM tagline	tagl	0.2 per tag per bank
Source tag CAM matchline	ml	0.6 per tag per bank
Global tag tagline	gtagl	2.1 per tag

Energy consumption of issue queue and register file components for 0.1 micron technology [5, 4, 2].

Bibliography

[1] J. Abella and A. González. Power-aware adaptive instruction queue and rename buffers. Technical Report UPC-DAC-2002-31, Universitat Politècnica Catalunya, 2002.

[2] Jaume Abella. Personal correspondence, 2002 - 2005.

[3] Jaume Abella, Ramon Canal, and Antonio González. Power- and complexity-aware issue queue designs. *IEEE Micro*, 23(5), 2003.

[4] Jaume Abella and Antonio González. On reducing register pressure and energy in multiple-banked register files. In *Proceedings of the 21st International Conference on Computer Design (ICCD-21)*, 2003.

[5] Jaume Abella and Antonio González. Power-aware adaptive issue queue and rename buffers. In *Proceedings of the 10th International Conference on High Performance Computing (HiPC)*, volume 2913 of *Lecture Notes in Computer Science*. Springer, 2003.

[6] Jaume Abella and Antonio González. Low-complexity distributed issue queue. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA-10)*, 2004.

[7] Aneesh Aggarwal and Manoj Franklin. Energy efficient asymmetrically ported register files. In *Proceedings of the 21st International Conference on Computer Design (ICCD-21)*, 2003.

[8] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[9] Haitham Akkary, Ravi Rajwar, and Srikanth T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36)*, 2003.

[10] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 2002.

[11] R. Iris Bahar and Srilatha Manne. Power and energy reduction via pipeline balancing. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA-28)*, 2001.

[12] Saisanthosh Balakrishnan and Gurindar S. Sohi. Exploiting value locality in physical register files. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36)*, 2003.

[13] Rajeev Balasubramonian, Sandhya Dwarkadas, and David H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO-34)*, 2001.

[14] Eric Borch, Srilatha Manne, Joel Emer, and Eric Tune. Loose loops sink chips. In *Proceedings of the 8th International Symposium on High Performance Computer Architecture (HPCA-8)*, 2002.

[15] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA-27)*, 2000.

[16] D. Burger and T. Austin. The simplescalar tool set, version 2.0. Technical Report TR1342, University of Wisconsin-Madison, 1997.

[17] J. Adam Butts and Gurindar S. Sohi. Use-based register caching with decoupled indexing. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA-31)*, 2004.

[18] Alper Buyuktosunoglu, David Albonesi, Stanley Schuster, and David Brooks. A circuit level implementation of an adaptive issue queue for power-aware microprocessors. In *Proceedings of the 11th Great Lakes Symposium on VLSI*, 2001.

[19] Alper Buyuktosunoglu, David H. Albonesi, Pradip Bose, Peter W. Cook, and Stanley E. Schuster. Tradeoffs in power-efficient issue queue design. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2002.

[20] Alper Buyuktosunoglu, Tejas Karkhanis, David H. Albonesi, and Pradip Bose. Energy efficient co-adaptive instruction fetch and issue. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA-31)*, 2004.

[21] Alper Buyuktosunoglu, Stanley Schuster, David Brooks, Pradip Bose, Peter Cook, and David Albonesi. An adaptive issue queue for reduced power at high performance. In *Proceedings of the Workshop on Power-Aware Computing Systems*, volume 2008 of *Lecture Notes in Computer Science*. Springer, 2000.

[22] Ramon Canal and Antonio Gonález. A low-complexity issue logic. In *Proceedings of the 14th International Conference on Supercomputing (ICS-14)*, 2000.

[23] Ramon Canal and Antonio González. Reducing the complexity of the issue logic. In *Proceedings of the 15th International Conference on Supercomputing (ICS-15)*, 2001.

[24] José-Lorenzo Cruz, Antonio González, Mateo Valero, and Nigel P. Topham. Multiple-banked register file architectures. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA-27)*, 2000.

[25] J. Emer. Ev8: The post-ultimate alpha. In *Keynote at the 10th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2001.

[26] Oğuz Ergin, Deniz Balkan, Kanad Ghose, and Dmitry Ponomarev. Register packing: Exploiting narrow-width operands for reducing register file pressure. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37)*, 2004.

[27] Oğuz Ergin, Deniz Balkan, Dmitry Ponomarev, and Kanad Ghose. Increasing processor performance through early register release. In *Proceedings of the 22nd International Conference on Computer Design (ICCD-22)*, 2004.

[28] Dan Ernst and Todd Austin. Efficient dynamic scheduling through tag elimination. In *Proceedings of the 29th International Symposium on Computer Architecture (ISCA-29)*, 2002.

[29] Dan Ernst, Andrew Hamel, and Todd Austin. Cyclone: A broadcast-free dynamic instruction scheduler with selective replay. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA-30)*, 2003.

[30] Brian Fields, Shai Rubin, and Rastislav Bodík. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA-28)*, 2001.

[31] Daniele Folegnani and Antonio González. Reducing power consumption of the issue logic. In *Proceedings of the Workshop on Complexity Effective Design (WCED) in conjunction with ISCA-27*, 2000.

[32] Daniele Folegnani and Antonio González. Energy-effective issue logic. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA-28)*, 2001.

[33] Soraya Ghiasi and Dirk Gurnwald. A comparison of two architectural power models. In *Proceedings of the Workshop on Power-Aware Computing Systems*, volume 2008 of *Lecture Notes in Computer Science*. Springer, 2000.

[34] A. González, J. González, and M. Valero. Virtual-physical registers. In *Proceedings of the 4th International Symposium on High Performance Computer Architecture (HPCA-4)*, 1998.

[35] Rubén González, Adrián Cristal, Daniel Ortega, Alexander Veidenbaum, and Mateo Valero. A content aware integer register file organization. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA-31)*, 2004.

[36] Stephen H. Gunther, Frank Binns, Douglas M. Carmean, and Jonathan C. Hall. Managing the impact of increasing microprocessor power consumption. *Intel Technology Journal*, Q1, 2001.

[37] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., second edition, 1996.

[38] Chung-Hsing Hsu, Ulrich Kremer, and Michael Hsiao. Compiler-directed dynamic frequency and voltage scheduling. In *Proceedings of the Workshop on Power-Aware Computing Systems*, volume 2008 of *Lecture Notes in Computer Science*. Springer, 2000.

[39] Chung-Hsing Hsu, Ulrich Kremer, and Michael Hsiao. Compiler-directed dynamic voltage/frequency scheduling for energy reduction in microprocessors. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2001.

[40] Jie S. Hu, N. Vijaykrishnan, and Mary Jane Irwin. Exploring wakeup-free instruction scheduling. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA-10)*, 2004.

[41] Zhigang Hu and Margaret Martonosi. Reducing register file power consumption by exploiting value lifetime. In *Proceedings of the Workshop on Complexity Effective Design (WCED) in conjunction with ISCA-27*, 2000.

[42] Michael Huang, Jose Renau, and Josep Torrellas. Energy-efficient hybrid wakeup logic. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2002.

[43] Nam Sung Kim, Todd Austin, Trevor Mudge, and Dirk Grunwald. Challenges for architectural level power modeling. *Power Aware Computing*, 2001.

[44] Nam Sung Kim, Krisztián Flautner, David Blaauw, and Trevor Mudge. Single-V_{DD} and single-V_T super-drowsy techniques for low-leakage high-performance instruction caches. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2004.

[45] Nam Sung Kim and Trevor Mudge. The microarchitecture of a low power register file. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2003.

[46] Nam Sung Kim and Trevor Mudge. Reducing register ports using delayed write-back queues and operand pre-fetch. In *Proceedings of the 17th International Conference on Supercomputing (ICS-17)*, 2003.

[47] Masaaki Kondo and Hiroshi Nakamura. A small, fast and low-power register file by bit-partitioning. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*, 2005.

[48] Gurhan Kucuk, Dmitry Ponomarev, Oğuz Ergin, and Kanad Ghose. Reducing reorder buffer complexity through selective operand caching. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2003.

[49] Gurhan Kucuk, Dmitry Ponomarev, and Kanad Ghose. Low-complexity reorder buffer architecture. In *Proceedings of the 16th International Conference on Supercomputing (ICS-16)*, 2002.

[50] Chingren Lee, Jenq Kurn Lee, TingTing Hwang, and Shi-Chun Tsai. Compiler optimization on instruction scheduling for low power. In *Proceedings of the 13th International Symposium on System Synthesis (ISSS-13)*, 2000.

[51] Mikko H. Lipasti, Brian R. Mestan, and Erika Gunadi. Physical register inlining. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA-31)*, 2004.

[52] Jack L. Lo, Sujay S. Parekh, Susan J. Eggers, Henry M. Levy, and Dean M. Tullsen. Software-directed register deallocation for simultaneous multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 10(9), 1999.

[53] Markus Lorenz, Rainer Leupers, Peter Marwedel, Thorsten Dräger, and Gerhard Fettweis. Low-energy DSP code generation using a genetic algorithm. In *Proceedings of the 19th International Conference on Computer Design (ICCD-19)*, 2001.

[54] Luis A. Lozano and Guang R. Gao. Exploiting short-lived variables in superscalar processors. In *Proceedings of the 28th International Symposium on Microarchitecture (MICRO-28)*, 1995.

[55] Grigorios Magklis, Michael L. Scott, Greg Semeraro, David H. Albonesi, and Steven Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA-30)*, 2003.

[56] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Pipeline gating: Speculation control for energy reduction. In *Proceedings of the 25th International Symposium on Computer Architecture (ISCA-25)*, 1998.

[57] Roberto Maro, Yu Bai, and R. Iris Bahar. Dynamically reconfiguring processor resources to reduce power consumption in high-performance processors. In *Proceedings of the Workshop on Power-Aware Computing Systems*, volume 2008 of *Lecture Notes in Computer Science*. Springer, 2000.

[58] Milo M. Martin, Amir Roth, and Charles N. Fischer. Exploiting dead value information. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO-30)*, 1997.

[59] José F. Martinez, Jose Renau, Michael C. Huang, Milos Prvulovic, and Josep Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO-35)*, 2002.

[60] Pierre Michaud and André Seznec. Data-flow prescheduling for large instruction windows in out-of-order processors. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7)*, 2001.

[61] Teresa Monreal, Antonio González, Mateo Valero, José González, and Victor Viñals. Delaying physical register allocation through virtual-physical registers. In *Proceedings of the 32nd International Symposium on Microarchitecture (MICRO-32)*, 1999.

[62] Teresa Monreal, Víctor Viñals, Antonio González, and Mateo Valero. Hardware schemes for early register release. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, 2002.

[63] Teresa Monreal, Victor Viñals, José González, Antonio González, and Mateo Valero. Late allocation and early release of physical registers. *IEEE Transactions on Computers*, 53(10), 2004.

[64] Mayan Moudgill, Keshav Pingali, and Stamatis Vassiliadis. Register renaming and dynamic speculation: an alternative approach. In *Proceedings of the 26th International Symposium on Microarchitecture (MICRO-26)*, 1993.

[65] Soner Önder and Rajiv Gupta. Superscalar execution with dynamic data forwarding. In *Proceedings of the 7th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 1998.

[66] Subbarao Palacharla. *Complexity-Effective Superscalar Processors*. PhD thesis, University of Wisconsin-Madison, 1998.

[67] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA-24)*, 1997.

[68] Il Park, Michael D. Powell, and T. N. Vijaykumar. Reducing register ports for higher speed and lower energy. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO-35)*, 2002.

[69] Matthew Postiff, David Greene, Steven Raasch, and Trevor Mudge. Integrating superscalar processor components to implement register caching. In *Proceedings of the 15th International Conference on Supercomputing (ICS-15)*, 2001.

[70] Steven E. Raasch, Nathan L. Binkert, and Steven K. Reinhardt. A scalable instruction queue design using dependence chains. In *Proceedings of the 29th International Symposium on Computer Architecture (ISCA-29)*, 2002.

[71] Guillermo Savransky, Ronny Ronen, and Antonio González. Lazy retirement: A power aware register management mechanism. In *Proceedings of the Workshop on Complexity Effective Design (WCED) in conjunction with ISCA-31*, 2004.

[72] André Seznec, Eric Toullec, and Olivier Rochecouste. Register write specialization register read specialization: A path to complexity-effective wide-issue superscalar processors. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO-35)*, 2002.

[73] Premkishore Shivakumar and Norman P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical Report 2001/2, Western Research Laboratory, 2001.

[74] Deszö Sima, Terence Fountain, and Péter Kacsuk. *Advanced Computer Architectures: A Design Space Approach*. Addison-Wesley Longman, 1997.

[75] Dezsö Sima. Superscalar instruction issue. *IEEE Micro*, 17(5), 1997.

[76] Dezsö Sima. The design space of register renaming techniques. *IEEE Micro*, 20(5), 2000.

[77] Michael D. Smith and Glenn Holloway. The Machine-SUIF documentation set. http://-www.eecs.harvard.edu/machsuif/software/software.html, 2000.

[78] The Standard Performance Evaluation Corporation (SPEC) CPU 2000. http://www.spec.org/cpu2000/.

[79] The SUIF compiler infrastructure. http://suif.stanford.edu/.

[80] Joel M. Tendler, Steve Dodson, and Steve Fields. POWER4 system microarchitecture. Technical report, IBM Server Group, 2001.

[81] Mark C. Toburen, Thomas M. Conte, and Matt Reilly. Instruction scheduling for low power dissipation in high performance microprocessors. Technical report, North Carolina State University, 1998.

[82] Robert M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1), 1967.

[83] Jessica H. Tseng and Krste Asanović. Banked multiported register files for high-frequency superscalar microprocessors. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA-30)*, 2003.

[84] Dean M. Tullsen and Brad Calder. Computing along the critical path. Technical report, University of California, San Diego, 1998.

[85] Steven Wallace and Nader Bagherzadeh. A scalable register file architecture for dynamically scheduled precessors. In *Proceedings of the 5th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 1996.

[86] W. Zhang, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, D. Duarte, and Y-F. Tsai. Exploiting VLIW schedule slacks for dynamic and leakage energy reduction. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO-34)*, 2001.

[87] Yan Zhang, Dharmesh Parikh, Karthik Sankaranarayanan, Kevin Skadron, and Mircea Stan. Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects. Technical Report CS-2003-05, The University of Virginia, 2003.

[88] V. Zyuban and P. Kogge. The energy complexity of register files. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 1998.