Banked Microarchitectures for Complexity-Effective Superscalar Microprocessors

by

Jessica Hui-Chun Tseng

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

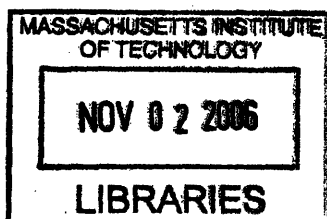at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2006
[June 2006]

Author ....
Department of Electrical Engineering and Computer Science
May 5, 2006

Certified by .... *j.w.jr. N.man* ....
Krste Asanović
Associate Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Banked Microarchitectures for Complexity-Effective Superscalar Microprocessors

by

Jessica Hui-Chun Tseng

Abstract

High performance superscalar microarchitectures exploit instruction-level parallelism (ILP) to improve processor performance by executing instructions out of program order and by speculating on branch instructions. Monolithic centralized structures with global communications, including issue windows and register files, are used to buffer in-flight instructions and to maintain machine state. These structures scale poorly to greater issue widths and deeper pipelines, as they must support simultaneous global accesses from all active instructions. The lack of scalability is exacerbated in future technologies, which have increasing global interconnect delay and a much greater emphasis on reducing both switching and leakage power. However, these fully orthogonal structures are over-engineered for typical use. Banked microarchitectures that consist of multiple interleaved banks of fewer ported cells can significantly reduce power, area, and latency of these structures. Although banked structures exhibit a minor performance penalty, significant reductions in delay and power can potentially be used to increase clock rate and lead to more complexity-effective designs. There are two main contributions in this thesis. First, a speculative control scheme is proposed to simplify the complicated control logic that is involved in managing a less-ported banked register file for high-frequency superscalar processors. Second, the RingScalar architecture, a complexity-effective out-of-order superscalar microarchitecture, based on a ring topology of banked structures, is introduced and evaluated.

Thesis Supervisor: Krste Asanović
Title: Associate Professor

Acknowledgments

First, I like to thank my advisor Krste Asanović for supporting me throughout this work and for his constant advice and encouragement. He is such a great inspiration to any student who is interested in computer architecture research. I treasure my learning experience under his invaluable guidance. I also like to thank Professor Arvind and Professor Srini Devadas for reading my thesis and sitting on my committee. I thank Professor Jacob White for his academic advice in pursuing my graduate study.

Thanks to Christine Chan, Abe McAllister, Godfrey Tan, Jaime Teevan, Doug De Couto, and Mark Hampton for their collaborations in various class projects. Thanks to Ronny Krashinsky and Mike Karcz-marek for TA'ing 6.823 with me. More thanks to Ken Barr, Chris Batten, Steve Gerding, Jae Lee, Rose Liu, Albert Ma, Heidi Pan, Michael Zhang, Seongmoo Heo, and Emmett Witchel for being the greatest teammates.

I thank Professor Dean Tullsen for providing and helping me with SMTSIM. I thank Xiaowei Shen for being my mentor when I was interning in IBM T. J. Watson Research Center. I thank Victor Zyuban for many interesting discussions in register file designs. I also thank Joel Emer for sharing his insightful experiences in microprocessor designs.

Thanks to Anne McCarthy for helping me settle in when I first joined the group. Thanks to Michael Vezza for providing IT support that saved me a great deal of time. Many thanks to Mary McDavitt for all her administrative help, encouragements, and moral supports during the last few years of my graduate work.

Thanks to all my friends who have made my life in MIT so enjoyable and memorable. Special thanks to Jing Song, Joy Cheng, and Carolyn Lee, for being such wonderful roommates and great friends during my study in MIT. I miss our late night snacks and conversations.

Last but not less, I like to thank Mom, Dad, David, and Will for their endless love, patience, support, and encouragements. Most of all, thank you for sharing this journey and believing in me.

5

Contents

9

List of Figures

11

List of Tables

Chapter 1

Introduction

Conventional superscalar microarchitectures [Yea96, Kes99] employ monolithic centralized structures with global communications, including issue windows, register files, and bypass networks. These structures scale poorly to greater issue widths and deeper pipelines, as they must support simultaneous global accesses from all active instructions. They also scale poorly to future technologies, which have increasing global interconnect delay and a much greater emphasis on reducing both switching and leakage power.

To address this problem, decentralized clustered microarchitectures have been proposed [SBV95, KF96, FCJV97, RJSS97, AG05], where microarchitectures are divided into disjoint clusters each containing local instruction windows, register files, and functional units. However, clustering adds significant control logic complexity to map instructions to clusters and to manage communication of values between clusters. The area and control complexity overhead of clustered architectures cannot be justified by the level of instruction parallelism in current applications [ZK01].

The other approach is to retain a centralized design but increase the efficiency of these fully orthogonal structures, which are over-engineered for typical usage. The challenge is in developing efficient microarchitectures with simple pipeline control algorithms that allow smaller, less-orthogonal structures to attain good performance without adding excessive complexity. In this thesis, I argue that a complexity-effective superscalar microprocessor can be realized by constructing centralized structures from multiple interleaved banks of lesser ported cells.

The research conducted for this thesis builds upon earlier work in banked register files [WB96, CGVT00, BDA01, PPV02], tag-elimination [EA02, KL03], and dependence-based scheduling [KF96, PJS97] and has

17

two main contributions:

- Developing a speculative control scheme for banked register files for high-frequency superscalar processors.

- Proposing a complexity-effective superscalar architecture that is based on a ring topology of banked microarchitectures.

1.1   Thesis Overview

This thesis consists of six chapters.

Chapter 2 provides background and motivation for this work. It reviews conventional superscalar microprocessors and shows that monolithic centralized structures with global communications such as register file, issue windows, and bypass network do not scale effectively with issue widths and deeper pipelines. Previous work in microarchitecture techniques that improve processor efficiency are examined, including clustered and banked architectures.

Chapter 3 describes the methodology and the baseline machine configuration used for this work. The SMTSIM simulator [Tul96] and the SPEC CINT2000 benchmark suite [Hen00] were used to evaluate the performance of various banked microarchitectures and their control logic. Detailed models of register file, instruction window, and control logic were added to SMTSIM. A baseline machine configuration is chosen based on a preliminary analysis of the characteristics of the workload.

Chapter 4 examines energy-efficient banked multiported register file designs. Provided that the number of simultaneous accesses to any bank is less than the number of ports on each bank, a banked regfile can provide the aggregate bandwidth needs of a superscalar machine with significantly reduced area compared to a fully multiported regfile. Custom layouts of the regfile were used to determine the physical characteristic of various banking designs. Earlier banked schemes that required complex control logic with pipeline stalls would have likely limited the cycle time of a high-frequency design. I present a speculative control scheme suitable for a deeply pipelined high-frequency dynamically scheduled processor which avoids pipeline stalls. The performance impact of regfile port conflict mis-speculations are evaluated and verified with results from both a cycle-accurate simulator and an analytical model.

18

Chapter 5 applies the banking techniques to the instruction issue window and proposes a complexity-effective banked architecture, RingScalar. RingScalar builds an N-way superscalar from N columns, where each column contains a portion of the instruction window, a bank of the register file, and an ALU. By exploiting dependency-based scheduling to place dependent instructions in adjacent columns, the bypass network is simplified into a ring connect, where a functional unit can only bypass values to the next functional unit. Furthermore, the fact that most instructions have only one outstanding operand when they enter the rename stage is exploited to use only one wakeup tag in the issue queue entries. This approach reduces the cost of broadcasting tag information across the window.

Finally, Chapter 6 concludes by summarizing the contributions of this thesis and suggesting future work.

Chapter 2

Background and Motivation

Out-of-order superscalar microarchitectures provide high single-thread performance, but at a significant cost in terms of area and power. This overhead is due to large centralized structures with global communications, including issue windows, register files, and bypass networks. This hardware dynamically extracts instruction-level parallelism (ILP) from a single instruction stream but scales poorly to greater issue widths and future technologies [PJS97], which have increasing global interconnect delay and rising leakage power. The advent of chip-scale multiprocessors, which integrate multiple cores per die, provides additional motivation to improve the area and power efficiency of each core. In this chapter, I provide an overview of superscalar processors and related microarchitecture techniques. Then, I argue that banked microarchitectures merit investigation for their high area and power efficiency which results in a more complexity-effective machine.

Section 2.1 reviews the basic concepts of a superscalar microprocessor and its major components. High performance superscalar designs raise the overall instruction thoughput by issuing multiple instructions in parallel and executing instructions out of program order. The pipeline structure of an out-of-order superscalar processor is illustrated and terminology is defined. I identify the reasons why orthogonal structures such as register files and instruction issue window are difficult to scale.

Section 2.2 discusses an alternative microarchitecture style, clustering, which aims to support greater issue widths at higher frequencies. Clustered architectures attempt to scale to larger issue widths by splitting the microarchitecture into distributed clusters, each containing a subset of register file, issue queue, and functional units [SBV95, KF96, FCJV97, RJSS97]. However, such schemes require complex control logic

21

to map instructions to clusters and to handle intercluster dependencies. Also, clustered designs tend to be less area efficient as they typically have low utilization of the aggregate resources [RF98].

Section 2.3 and Section 2.4 examine previous research in microarchitecture techniques that alleviate the scalability and efficiency problem of register files and instruction windows. Section 2.5 summarizes the main arguments made in this chapter.

2.1 Superscalar Microprocessors

Superscalar microprocessors started appearing in the late 1980s and early 1990s with the increasing popularity of RISC processors and the growing availability of transistor resources. Recent commercial general-propose processors such as Silicon Graphics's MIPS R10000 [Yea96], Digital's 21264 Alpha [Kes99], IBM's Power4 [TDF+01], Intel's P6-based processors, the Pentium 4 [HSU+01], as well as AMD's Athlon [AMD99] and Optero [KMAC03] are all superscalar processors.

2.1.1 In-order vs. Out-of-order

Superscalar microarchitectures increase processor performance by issuing multiple instructions simultaneously to exploit the instruction-level parallelism (ILP) present in applications. A processor that can execute N instructions in parallel is an N-way machine. The ability to dynamically issue varying numbers of instructions per clock cycle differentiates the superscalar from the VLIW (very long instruction word) design. In a VLIW design, the compiler schedules instructions statically. An in-order superscalar issues instructions dynamically depending on the dynamic detection of hazards, but always in program order. Out-of-order designs issue instructions potentially out of program order. Out-of-order execution machines require additional hardware to manage the precise architectural state of in-flight instructions, but reduce processor stalls from data dependencies. For example, in an in-order core, the pipeline stalls when the next waiting instruction is the consumer of data that is not yet available. In contrast, out-of-order processors perform future work by issuing other instructions that do not have an outstanding data dependency into the pipeline.

To evaluate the performance benefits of superscalar, Figure 2-1 shows the Instruction-committed-per-cycle (IPC) comparisons of different machines across the SPEC2000 CINT benchmark suite by using the methodology described in the next chapter. The simulated data indicates an approximately 40% improvement from a single-issue to a quad-issue in-order machine and another 40% improvement from a four-way
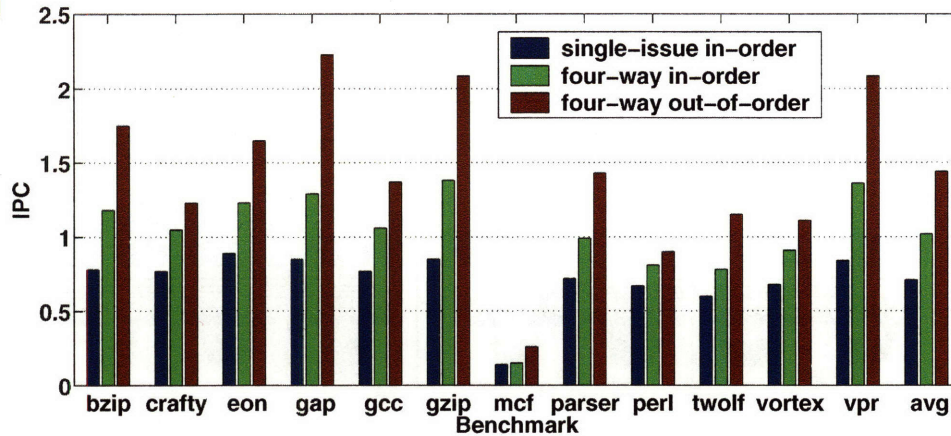
Figure 2-1: IPCs for single-issue in-order, four-way in-order, and four-way out-of-order superscalar machines.

in-order to a four-way out-of-order superscalar machine.

However, Figure 2-2 shows that as the issue width quadruples, demand on the number of register file port increases by a factor of four and the bypass network grows substantially. For out-of-order superscalars, the number of register file entries must increase to support removal of artificial data dependencies by register renaming. An instruction issue window is also required to manage and schedule in-flight instructions out of program order. These highly centralized and orthogonal structures such as the register file, instruction issue window, and bypass network are typically very inefficient. Therefore, the focus of this thesis is finding techniques to improve the efficiency of high performance out-of-order superscalar processors, while retaining their performance advantages.

2.1.2 Terminology

Throughout the thesis, the following terminology will be used to describe dynamically scheduled superscalar execution cores.

- **Fetch** is the act of retrieving instructions from the instruction cache.

- **Decode** is the process of decoding instruction opcodes and operands.

- **Rename** is the act of mapping an instruction's architectural registers to the physical registers. Renaming eliminates artificial write-after-read and write-after-write data dependency hazards [HP02].

23
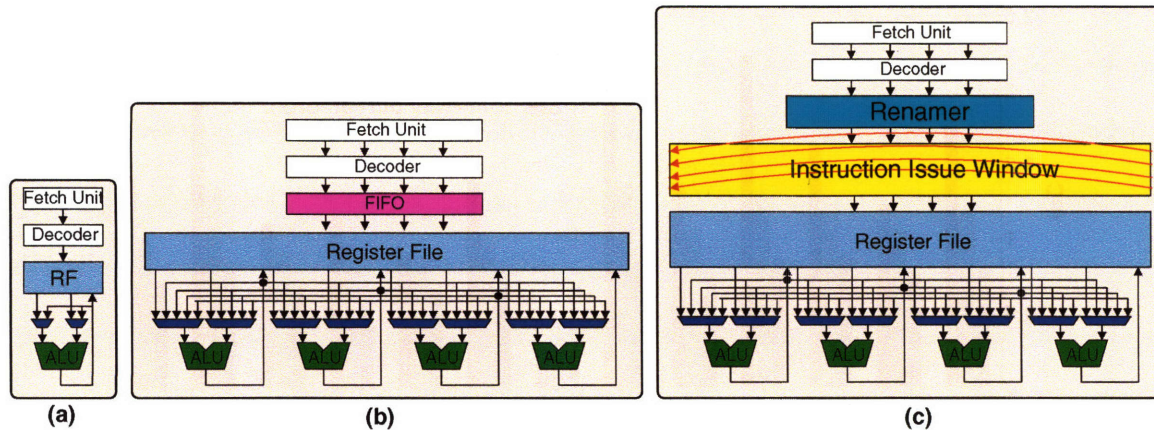
Figure 2-2: Microarchitectures of (a) single-issue in-order, (b) four-way in-order, and (c) four-way out-of-order superscalar machines.

- **Dispatch** is the process of moving the renamed instructions into the scheduling hardware, such as an instruction issue window.

- **Issue** is the process of sending ready instructions from the instruction issue window into the execution units.

- **Write Back** is the act of writing results back to the physical register file.

- **Complete** is the act of instructions leaving the pipeline after write back.

- **Commit** is the process of retiring completed instructions. The commit logic updates the architectural state and releases resources such as entries in the reorder buffer (ROB), memory queue, and register file.

2.1.3   Out-of-order Execution Pipeline

A superscalar processor builds on a logically pipelined design, where each pipeline stage is in charge of one of multiple tasks that are needed to complete each instruction. Figure 2-3 shows an example of a four-way out-of-order superscalar processor execution pipeline structure. Instructions are first fetched, decoded, and renamed. Instructions are then dispatched to the issue window and wait until both operands are available. The instruction issue window pipeline stage contains the critical wakeup-select loop [PJS97], where the wakeup phase is used to update operand readiness and the select phase picks a subset of the ready instructions

24

| Fetch | Decode | Rename | Issue Wakeup Select | Read Bypass | Execute | Writeback | Commit |

Figure 2-3: Four-way out-of-order superscalar execution pipeline.

to issue. Once a single-cycle instruction is selected, its result tag is immediately broadcast to the instruction issue window in the next wakeup phase to allow back-to-back issue of dependent instructions, even though the selected instruction will not produce its result for several cycles. Before execution, register values are read from either the register file or the bypass network. After execution in the functional units, the results are written back to the register file and the instruction is committed in program order to maintain precise architectural state.

2.1.4 Register File

The multiported register file, or regfile, provides buffered communication of register values between producer and consumer instructions. With the deeper pipeline speculation and higher instruction-level parallelism (ILP) of more aggressive out-of-order superscalar processor designs, both the number of ports and the number of required registers increase. These increased requirements cause the area of a conventional multiported regfile to grow more than quadratically with issue width [ZK98]. The trend towards simultaneous multithreading (SMT) [TEL95] further increases register count as separate architectural registers are needed for each thread. For example, the proposed eight-issue Alpha 21464 design had a regfile that occupied over five times the area of the 64 KB primary data cache [Pre02].

A register file is composed of SRAM like storage cells, address decoders, wordline drivers, column circuitry, multiplexing circuitry, and interconnects [ZK98, RDK$^+$00, SJ01, BPN03]. Figure 2-4 shows a 32×32-bit regfile with two single-ended read ports and one differential write port. The storage array is the main component and the cell size is constrained by the number of wires (bitline and wordlines) and the cross-coupled inverters. We can express the area of a multiported storage cell with single-ended reads and differential writes in a closed form, Equation 2.1. h_{inv} is the height of the coupled-inverters layout, w_{inv} is the width of the coupled-inverter layout, W is each wire-track space, N_r is the number of read ports, and N_w is the number of write ports.

25
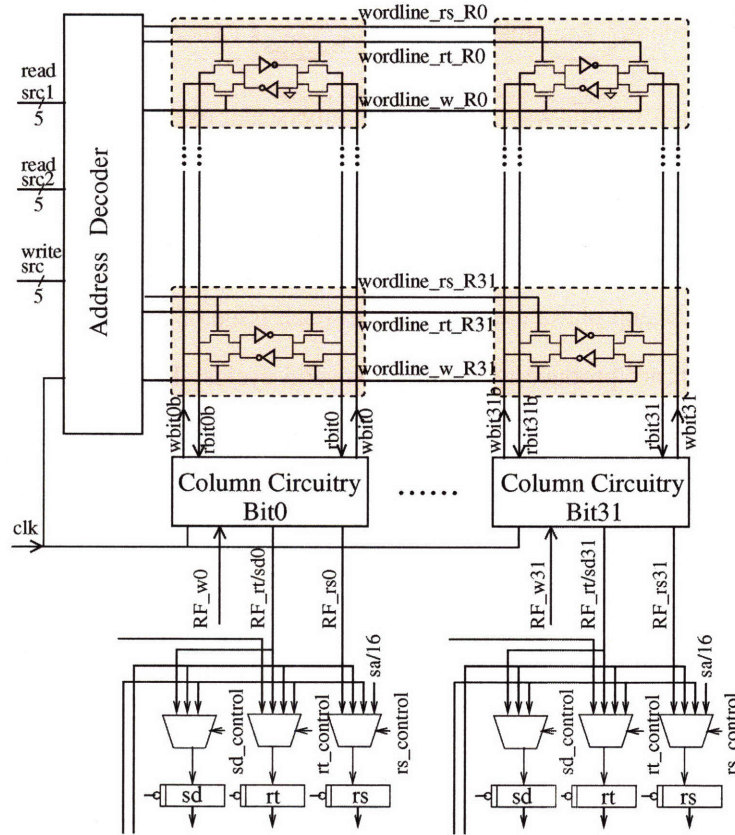
Figure 2-4: A 32×32 regfile structure with two read-port and one write-port.

$$area_{storage} = (h_{inv} + W(N_r + N_w)) \times (w_{inv} + WN_r + 2WN_w) \qquad (2.1)$$

Both Figure 2-4 and Equation 2.1 indicate that the area of a conventional regfile increases super-linearly with the number of ports. For this reason, many architects have explored alternative designs for implementing a large and fast multiported register file.

One approach, used in the Alpha 21264 [Kes99] and 21464 [Pre02] designs, divides the functional units among two clusters and provides a copy of all registers in each cluster. This design halves the number of read ports required on each copy of the regfile, but requires the same number of write ports on both regfiles to allow values produced in one cluster to be made available in the second cluster. This approach isn't used to reduce area but to reduce read latency.

2.1.5 Instruction Issue Window

The instruction issue window or issue queue is where instructions wait to be scheduled to an appropriate execution unit. The issue window dynamically schedules instructions into the pipeline to exploit instruction-level parallelism to improve processor performance. Because the critical wakeup-select loop is part of the instruction issue window and its timing usually helps determine a processor's clock frequency, it is hard to scale this structure to greater issue widths [PJS97]. Moreover, studies have shown that the scheduler logic consumes a large portion of a processor's power (e.g. around 18% of total chip power of the Alpha 21264 [EA02]).

There are several styles in implementing a dynamic scheduler, but most designs employ a wakeup circuit to check the read-after-write data hazard and a select arbiter to choose instructions for issue. One approach is to store the direct source tags of each waiting instruction and dynamically update its readiness by matching the source tags to the result tags of issued instructions. As shown in Figure 2-5, each result tag passes through one of N wakeup ports and is driven across the entire window, with each entry having two comparators, leading to a fanout of $2E$, where E is the number of entries. The direct source-tag scheduler requires $N\lceil\log_2(X)\rceil$ wires to broadcast N result tags each cycle, where X is the number of regfile entries, and a total of $E \times 2N\lceil\log_2(X)\rceil$ bit-comparators. This scheme is shown to scale well with larger number of registers but not with greater issue width.
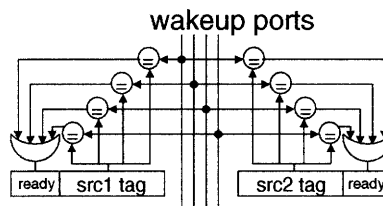
Figure 2-5: Wakeup circuitry of a scheduler.

Another variation of the direct source-tag approach is to one-hot encode the source tags as in the Alpha 21264 [FF98] processor. Each entry of the issue queue contains as many number of bits as the number of physical regfile entries, X. This bit array represents the data dependencies of each instruction and is the logical OR of two one-hot encoded sources. Again, the scheduler updates the readiness of instructions by matching the source field to destination registers of issued instructions each cycle. To reduce the number of wakeup ports, result tags (one-hot encoded) of issued instructions are ORed together. Since there is only
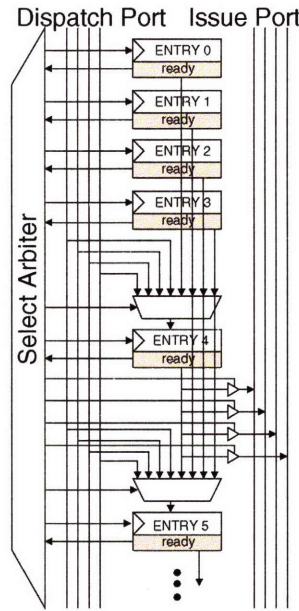
Figure 2-6: Latch-based compacting instruction issue queue.

one set of comparators for the source field, fanout for the wakeup port is also reduced from $2E$ to E. The one-hot encoding scheme requires only a single wakeup port but a larger storage array for issue window ($E \times X$ instead of $E \times 2\lceil \log_2(X) \rceil$). This design requires X wires to signal data completion and a total of EX bit-comparators.

The size of an issue window that uses one-hot encoding grows linearly with the size of regfile and becomes impractical for designs with a large number of registers. To improve the efficiency, Intel's Pentium 4 processor [HSU+01] stores the instruction dependencies in terms of one-hot encoded scheduler entry numbers instead of register numbers. The issue window is organized as a $E \times E$ bit matrix. As instructions are issued, they send the one-hot encoding that correspond to their issue queue entry numbers. This scheme requires E number of wires with a fanout of E to broadcast the entry numbers of issued instructions and a total of E^2 bit-comparators. The complexity reductions of the scheduler, however, comes at the cost of an additional future-file like structure and the control logic that maps the architectural register numbers to scheduler entry numbers.

Many issue queue designs keep instructions in age-order because it simplifies pipeline cleanup after exceptions or branch mispredictions and can improve performance over randomly assigned order. Figure 2-6 shows a latch-based compacting instruction queue design where the fixed-priority select arbiter picks the oldest ready instruction for issue. The age ordering of instructions is preserved by compacting out a
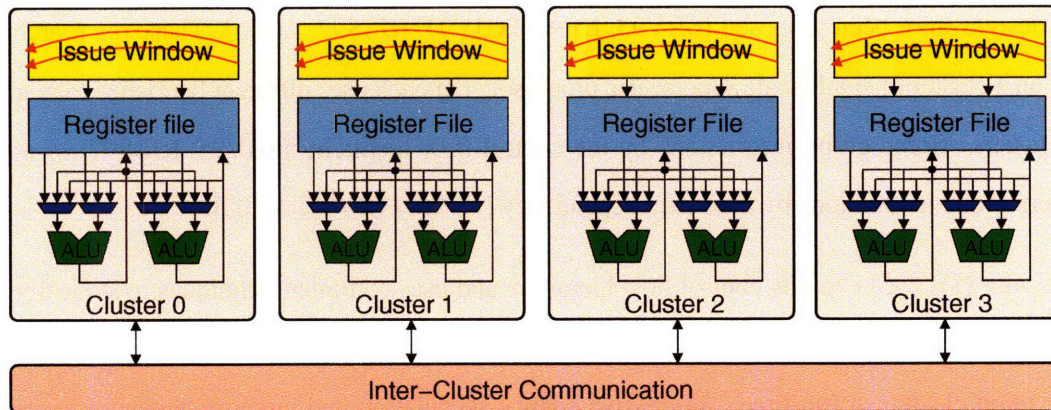
Figure 2-7: Clustered architecture.

completed instruction from the window, each entry below the hole retains its value, each entry at or above the hole copies the entry immediately above to squeeze out the hole, while the previous highest entry copies from the dispatch port if there's a new dispatch. Each entry has a single input port which is connected to a $2N$ fan-in mux, feeding from N dispatch ports and output ports of entries above. The dispatch ports are used to move the newly decoded instruction into the issue window. Also, each entry has one output port with a fanout of $2N$, N to issue ports and N to input port of entries below. After instructions are selected, they are sent to the issue ports for further pipeline execution. The wide input mux and the large fanout of latch-based compacting issue queues makes it hard to scale to greater issue width.

Dynamic instruction schedulers are a central component in superscalar microprocessors and have received much attention in the literature [PJS97, BAB+02, RBR02, PKE+03]. Previous techniques that try to prevent the scheduler from becoming the critical path will be reviewed and discussed in Section 2.4.

2.2 Clustered Architectures

Clustered architectures divide the microarchitecture into disjoint clusters each containing local issue windows, register files, and functional units as shown in Figure 2-7. Inter-cluster communication is required when a value is needed from a different cluster. Clustering has the potential to scale with larger issue widths and at high clock frequencies but its performance can be affected by poor workload balancing and high inter-cluster communication penalties. Therefore, a critical matter in the design of such systems is the heuristics used to map instructions to clusters.

To address the delay and power scaling problem of superscalar machines, various forms of decentralized

29

clustered designs have been proposed [SBV95, KF96, FCJV97, RJSS97, AG05]. These microarchitectures are based on a cluster of small-scale superscalar processors but each uses different heuristics to map instructions to clusters. Some group instructions by the control flow hierarchy [SBV95, RJSS97] of a program while others group instructions by their data dependency [KF96, FCJV97, AG05].

Multiscalar [SBV95] exploits control flow hierarchy and uses compiler techniques to statically divide a single program into a collection of tasks. Each task is then dynamically scheduled and assigned to one of many execution clusters at run-time. Aggressive control speculation and memory dependence speculation are implemented to allow parallel execution of multiple tasks. To maintain a sequential appearance, each task is retired in program order. This scheme requires extensive modification to the compiler and the performance depends heavily on the ability to execute tasks in parallel. On the other hand, Trace processors [RJSS97] are software invisible and build traces dynamically as the program executes. This approach, however, requires a large centralized cache to store the traces and the algorithm used to delineate traces strongly influences the overall performance. Both Multiscalar and Trace processors attempt to reduce inter-cluster communication by localizing program segments to an individual cluster but rely largely on control and value predictions for parallel execution.

Data dependencies dictate the amount of communication between instructions. PEWs [KF96] and Multicluster [FCJV97] assigns dependent instructions to the same cluster as much as possible to minimize inter-cluster communication and these assignments are determined at decode time. The performance is comparable to a centralized design but decreases as the number of clusters increases. This is because such an algorithm has poor load balancing and cannot effectively utilize a large number of clusters. Multicluster suggests the possibilities of using complier techniques to increase utilization by performing code optimization and code scheduling. Instead, Abella and Gonzalez [AG05] proposed a scheme that inherently distributes the workload across all the clusters by placing consumer instructions in the cluster next to the cluster that contains the producer instructions. To minimize the cost of such next-neighbor cluster communication, the processor is laid out in a ring configuration so that the results of a cluster can be forwarded to the neighbor cluster with a very short latency as shown in Figure 2-8. Each partition of the register file can be read only from its cluster but can be written only from the previous neighboring cluster. Nevertheless, this design still requires long latency inter-cluster communications to move missing operands to appropriate clusters. I use a variant of this ring topology in the context of a centralized microarchitecture to reduce complexity.

30
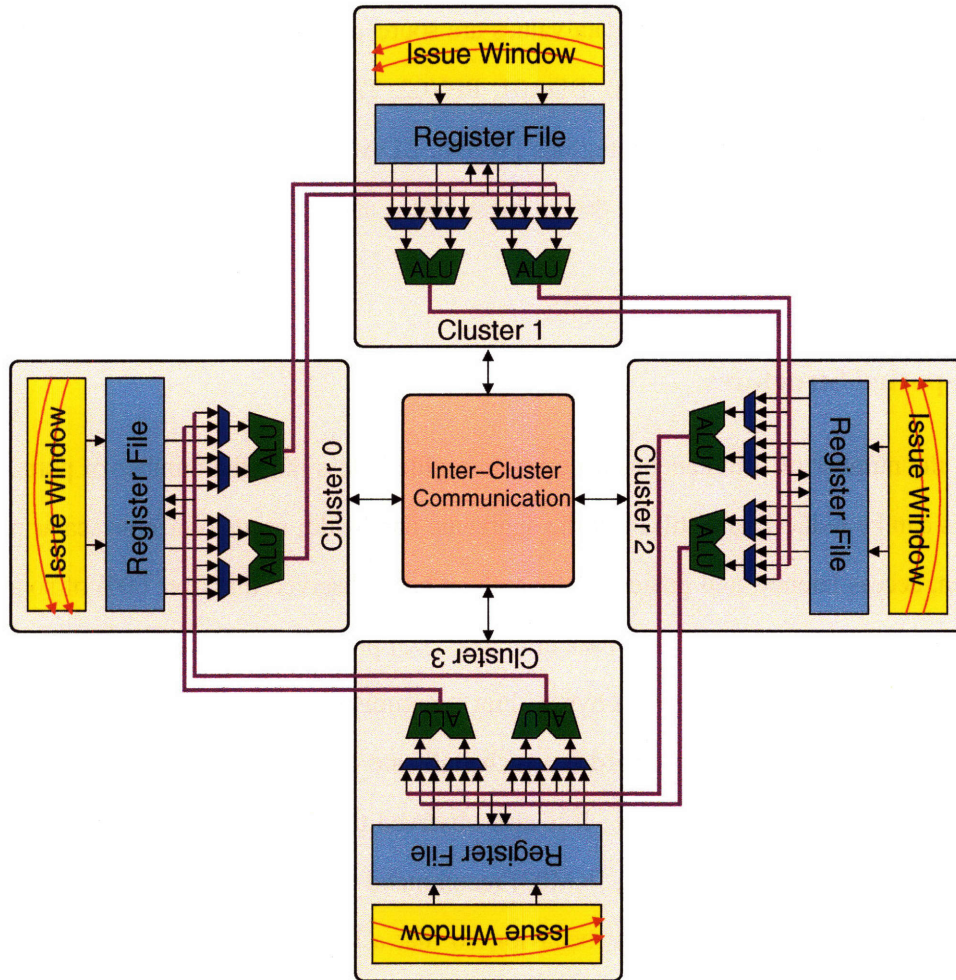
Figure 2-8: Ring clustered architecture.

Clustered architectures have the potential to scale to larger issue widths by splitting the microarchitecture into distributed clusters. However, the primary disadvantages are the complexity of inter-cluster control logic and the complexity of mapping instructions to clusters. Clustered designs also require additional area to achieve performance similar to a centralized architecture for moderate size cores [ZK01]. Since our goal is to build a reduced complexity medium-size core, rather than an aggregation of multiple medium-sized cores to build a large core, the banking approach is a more suitable alternative.

2.3 Efficient Register File Designs

A variety of techniques have been proposed to reduce the complexity of a multiported regfile by exploring different types of register access behaviors. For example, it is well known that a conventional regfile is

over-engineered for typical usage. Many data values are sourced from the bypass network, leaving regfile read ports underutilized [WB96, CGVT00, BDA01, PPV02, TA03]. This motivates the designs of less-ported [PPV02, KM03] and multibanked regfiles [WB96, BDA01]. Additionally, localities in access and localities in value have been exploited to design hierarchical and simplified regfile structures [CGVT00, BTME02, JRB+98, BS03, AF03].

2.3.1 Less-ported Structures

Provided that the number of simultaneous accesses to regfile is less than the number of ports, the number of regfile ports can be reduced and still meet the aggregate bandwidth needs of a superscalar machine with savings in area, access latency, and power. Many authors have observed that over 60% of source operands are sourced from the bypass network and Park et al. [PPV02] proposed a bypass-hint scheme that reduces register read port contention. However, this bypass-hint is optimistic and can be incorrect. Pipeline stalls are required when read ports are oversubscribed but such stalls are difficult to implement in a high frequency pipeline without compromising cycle time. It is also observed that in over 50% cycles, all read ports and write ports are idle [HBHA02]. Kim et al. [KM03] recommend pre-fetching operands and adding delayed write-back queues to smooth out the burst behavior, so that a less-ported regfile can be used to provide the aggregate bandwidth, but this adds the cost of extending the bypass network. Nevertheless, both these proposals add complexity to the already timing critical wakeup-select loop, because the select logic still has to select no more instruction than the number of available read ports after considering the bypass hint bits [PPV02] or the prefetch flags [KM03].

2.3.2 Multibanked Microarchitecture

Similar performance can be reached with even fewer ports on each storage cell if the regfile is built from multiple interleaved banks [WB96, BDA01] (Figure 2-9). Such multibanked register file designs have been shown to provide sufficient bandwidth for a superscalar machine with significantly reduced area compared to a fully multiported regfile [WB96, BDA01]. The challenge with this approach is managing the complexity and added latency of the control logic needed to handle read and write bank conflicts and the mapping of register ports to functional units. Previous designs all have complex control structures that would likely limit cycle time and add to design complexity.
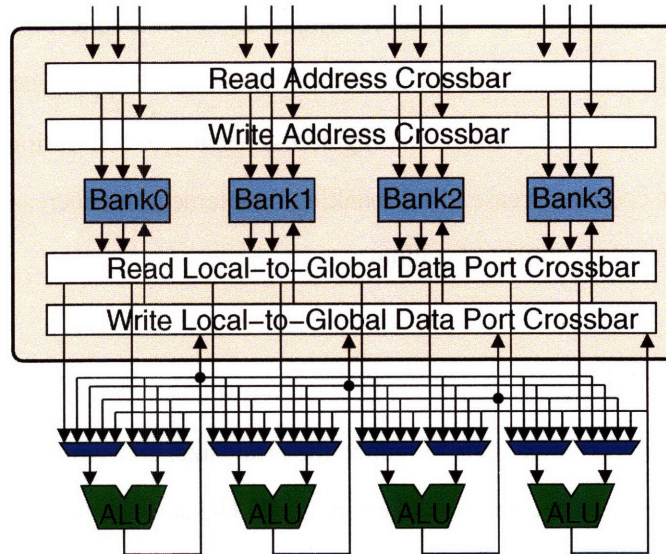
Figure 2-9: Multibanked regfile structure.

Wallace [WB96] describes a banking scheme that uses the bypass network to reduce unnecessary read port contention but no description of the bypass check or read conflict resolution logic is given. Also, write conflicts are handled by delaying physical register allocation until writeback, at which point registers are mapped to non-conflicting banks. The primary motivation for this delayed allocation was to limit the size of the physical register file, but this can lead to a deadlock situation requiring a complex recovery scheme [WB96].

The scheme presented in [BDA01] handles read bank conflicts by only scheduling groups of instructions without conflicts. This reduces the IPC penalty but adds significant logic into the critical wakeup-select loop. The authors also assume that bypassability can be determined during wakeup, but do not detail the mechanism used [BDA01]. A design with single-ported read banks is evaluated. However, this requires more complex issue logic to allow instructions where both operands originate from the same bank to be issued across two successive bank read cycles. Also, write port conflicts are handled by buffering conflicting writes, which increases the size of the bypass network. Functional unit pipelines must also be stalled when the write buffers are full.

2.3.3 Register Caching

Localities of access have been used to design cache structures and can be extended to regfile designs. In [CGVT00, BTME02], registers are cached to reduce average access latency. The disadvantage of reg-

ister caching is that it can add considerable control complexity to an architecture, as register caches have much worse locality than conventional data caches and determining the appropriate values to cache is non-trivial. Register caching is motivated by the increasing access penalty of conventional multiported structures as port counts and register counts increase. Multibanking counteracts this increase and reduces total area, independent of the presence of exploitable localities.

2.3.4 Asymmetric Structure

As an alternative to less-ported storage cells to reduce the array size, an asymmetrically ported register file structure is proposed in [AF03] to decrease the wordline load. The author explores the fact that many of the register values are small and require only a small number of bits for representation to reduce the number of ports required for the most significant bits. The most significant bits of a regfile are less ported while the rest are fully ported. Again, this scheme involves additional complexity for the select logic and extra hardware that only reduces the wordline load. This asymmetric regfile structure still has problems with the scalability of the number of registers and the number of ports for the less significant bits.

2.3.5 Content Awareness

Clustering and banking approaches enable the implementations of regfiles with a large number of registers. Alternatively, localities of register values have been used to design content-aware register files to reduce the number of registers required [JRB+98, BS03]. Both Jourdan [JRB+98] and Balakrishnan [BS03] propose a regfile design that only allocates a single physical register for each unique value. This scheme reduces the number of required physical registers by eliminating value duplication. However, reusing a physical register for different logical registers necessitates a large amount of additional hardware and complicated control logic in both the critical wakeup-select loop and regfile write backs.

2.3.6 Other Related Work on Reduced Complexity Register Files

The aforementioned work has focused on the design of a high bandwidth register file for dynamically sched-uled superscalar processors with a single logical register file. Other work has examined the use of partitioned register files made visible to software. The SPARC architecture [WG94] has overlapping register windows where software explicitly switches between sets of registers. In-order superscalar implementations of the

UltraSPARC exploit the fact that only one register window is visible to implement a dense multiported structure [TJS95]. Clustered VLIW machines make the presence of multiple register file banks visible to software, and the compiler is responsible for mapping instructions to clusters [Fis83]. Vector machines have also long been designed with interleaved register file banks that exploit the regular access patterns of vector instructions to provide high bandwidth with few conflicts [Cor89, DEC90].

2.4 Efficient Instruction Issue Window Designs

Many researchers identify the issue window's wakeup and select logic as likely one of the most critical timing and energy bottlenecks for future superscalar processor designs [PJS97, BAB$^+$02, RBR02, PKE$^+$03]. Hence, several proposals have attempted to address this problem by improving its efficiency. The fact that issue window source tags are underutilized because many operands are ready before an instruction is dispatched into the window has inspired the tag-elimination [EA02], half-price architecture [KL03], and banked issue queue [BAB$^+$02] designs. These schemes reduce the electric loading of the wakeup ports and the number of comparators by keeping the number of tag checks to minimum. Others exploit the fact that it is unnecessary to perform tag checks on a consumer instruction prior to the issuing of its producer because a consumer instruction cannot possibly be issued before its producer. This motivates designs that group instructions in the same dependency chain to reduce wakeup complexity [KF96, PJS97, RBR02, EHA03].

2.4.1 Tag Elimination

Several authors have noted that instructions often enter the instruction window with one or more operands ready; only 10% to 20% of all dynamic instructions wait on two operands [EA02, BAB$^+$02, KL03]. Ernst and Austin [EA02] introduce a last-tag speculation technique to reduce tag comparison requirements for instructions with multiple operands in flight. Tag comparison is only performed for operands predicted to be last-arriving, which need only a single tag matching logic per issue queue entry. If a misprediction is discovered in the register read stage, the processor must flush and restart the pipeline. In comparison to the conventional scheduler, this scheme halves the number of tag comparisons but still requires the N wakeup port, N dispatch ports, and N issue ports. Also, it is doubtful if the saving can justify the cost in implementing a last-tag predictor and the extra control complexity in validation/recovery of last-tag prediction.

As an alternative, the Half-Price architecture [KL03] avoids last-tag misprediction by providing tag

comparison circuitry to both operands as in the conventional design. Instructions can only be issued into the pipeline if all dependencies are met. However, only half of the operands are directly wired to the wakeup bus to reduce load capacitance. To deliver data completions to the other half of the operands, a sequential wakeup mechanism is proposed to latch and rebroadcast the tag in a second broadcast path in the next clock cycle. Similar to last-tag speculation, a predictor is used to predict the timing critical operand that is then placed in the same-cycle-broadcast half of the issue window. The Half-Price architecture reduces the average latency of wakeup circuitry but still requires the same amount of hardware support and complexity as a conventional scheduler.

2.4.2 Banked Configuration

Buyuktosunoglu et al. [BAB$^+$02] proposed a banked issue queue structure for instructions with one or less non-ready operands and a separate queue to handle instructions with two non-ready operands as shown in Figure 2-10. Each instruction is steered to the bank corresponding to the ID number of the unavailable source register. This scheme reduces the power dissipation by halving the number of comparators in the banked issue queues and allowing only one bank to be activated per tag-broadcast. However, it doesn't scale well with greater issue width as each window requires N dispatch ports, N wakeup ports, and N issue ports to support the seamless flexibility of a monolithic scheduler. The large dispatching crossbar and issuing multiplexing circuits can also add considerable overhead. Plus, a standard two-waiting queue is still required in this scheme to schedule instructions that have neither source operand available.
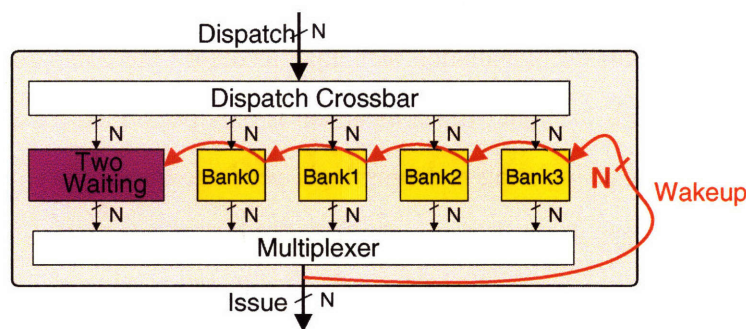
Figure 2-10: Banked issue queue organization.

2.4.3 Pipeline Window

Raasch et al. [RBR02] pipeline the issue queue by dividing it into small window segments to reduce cycle time. The processor dynamically promotes instructions from one segment to the next until they reach the final stage. This design eliminates the conventional tag broadcast and simplifies the select-logic because it keeps track of the dependency chains and only issues the instructions that reach the final segment. However, this scheme involves complex control logic, with considerable area and power overhead to determine the timing of each instruction. Extra pipeline stages also increase the branch misprediction latency and potential race conditions between dependency chains can cause deadlocks.

2.4.4 Scoreboard Scheduler

Conversely, the Cyclone scheduler [EHA03] estimates the issue time of each instruction based on its dependencies before dispatching. It uses a centralized scoreboard structure to update any timing variation. The processor checks the availability of operands before executing the instructions to avoid deadlocks and pipeline flushes. The instruction is simply replayed when it is scheduled incorrectly. Nonetheless, it can waste a great deal of energy to keep track of dependency chains by constantly moving instructions among the queues.

2.4.5 Distributed Scheduler

Several other schemes for dependence-based scheduling [KF96, PJS97] attempt to steer up to N dependent instructions from a dispatch buffer to one of N distributed instruction windows. The motivation is to localize wakeup processing in each window and to avoid considering dependent instructions in the select process. However, the simple FIFO-based approach presented by Palacharla et al. [PJS97] still requires that N dependent instructions can be steered into the tail of a FIFO at dispatch time as shown in Figure 2-11. This results in an $N \times N^2$ interconnect crossbar to allow any of the N dispatched instructions to connect to any of the N dispatch ports on any of the N FIFOs.

2.4.6 Other Related

To simplify the select-logic, some designers use separate groups of reservation stations [Tom67] for each functional units, such as in various IBM Power architecture implementations [TDF+01]. Nevertheless,
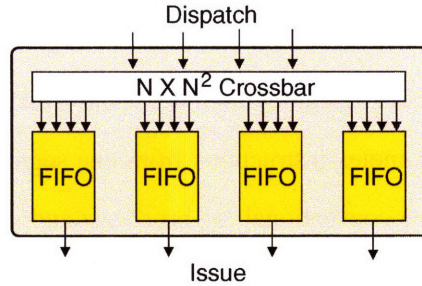
Figure 2-11: Distributed FIFO structured issue window.

Figure 2-12 shows that the result tag broadcast still needs to be sent to all the issue queues, and the dispatch network can become large if multiple instructions can be sent to one functional unit. Ponomarev et al. [PKE+03] decreases the energy consumption of issue queue by using circuit techniques but doesn't improve the latency.
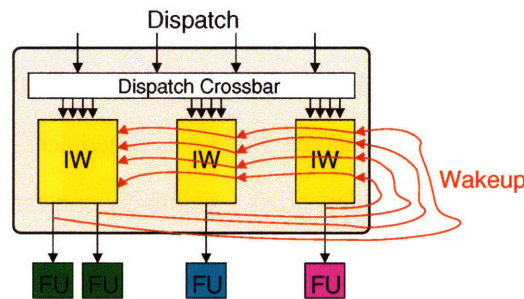
Figure 2-12: Reservation station style issue window design.

2.5   Motivation for Banked Microarchitectures

As reviewed above, several proposals have attempted to reduce the cost of one component of a superscalar architecture (e.g., just the register file or just the issue window), but often with a large increase in overall pipeline control complexity or possibly needing compensating enhancements to other portions of the machine (e.g., extending the bypass network to forward values queued in write buffers ahead of limited regfile write ports). Also, many schemes optimize only latency not area or power. This motivates the work in this thesis to develop a new out-of-order superscalar microarchitecture that simplifies all the major components in the instruction flow to increase area and power efficiency without excessive pipeline control complexity.

Clustering is a common approach to resolve the scalability problem of highly centralized superscalar
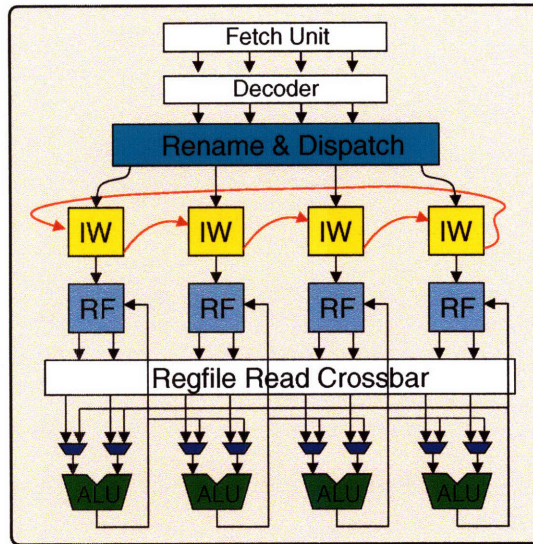
Figure 2-13: A multibanked architecture design.

structures. Although decentralized microarchitectures reduce intra-cluster latency and have the potential to scale to larger issue widths, they require complex logic to map instructions to clusters and to manage communication of values between clusters. Clustered architectures also tend to be less area efficient as they typically have low utilization of the aggregate resources [RF98]. I believe these are the reasons that few clustered processors have been implemented commercially. Alternatively, banked microarchitectures have attract many researchers' attention because of their ability to reduce area, power, and access time. Banking is more effective than clustering for moderate scale machines. In a cluster architecture, each cluster contains one slice of each major microarchitectural component (issue window, register file, functional units, and bypass network) placed close to each other. Multiple clusters are then connected with some lower-bandwidth inter-cluster communication mechanism. In a banked architecture, the slices within each component are placed next to each other, then components are interconnected in much the same way as in a monolithic design. This can be seen in Figure 2-13, where the regfile is a monolithic block divided into banks, where any column's ALU can read data from any bank. Another example is that all ALUs of a design with banked structures would be co-located in the same datapath to give very fast and low energy bypassing.

The contribution of this thesis is to improve previous work on banked regfile with a simpler and faster control scheme, and to extend banking to other parts of processor designs to reduced complexity of medium sized cores.

Chapter 3

Methodology

Microarchitecture designs are compared on the basis of their performance, power, area, and design complexity. The potential trade-off of these characteristics can be analyzed by careful study and comparison of circuits for each component individually. However, the impact of microarchitecture on overall processor performance is difficult to determine analytically due to the complex interactions between pipeline stages. In this chapter, I describe the common simulation methodology used to evaluate design alternatives. I also discuss the choice of workload and system parameters. Section 3.1 describes the general simulation framework, the simulation flow, and the functional blocks. Section 3.2 presents the benchmark suite and the sampling method. The machine configurations of idealized baseline processors are presented in Section 3.3. Section 3.4 summarizes the performance evaluation methodology.

3.1   Simulation framework

To characterize the behavior of banked microarchitectures, we extensively modified an existing superscalar processor simulator, SMTSIM [Tul96], to include detailed models of various banked microarchitectures and pipeline control schemes. These modifications included changes to the Rename, Dispatch, Select, Issue, Regfile, and Bypass stages of the processor pipeline.

SMTSIM [Tul96] is an instruction-level simulator that provides a cycle-accurate model of a pipelined out-of-order superscalar processor with simultaneous multithreading. SMTSIM is written in the C programming language and executes the Alpha instruction set. Modifications are made to SMTSIM to represent the idealized baseline machine, which uses the same general structure as the MIPS R10K [Yea96] and Alpha

41
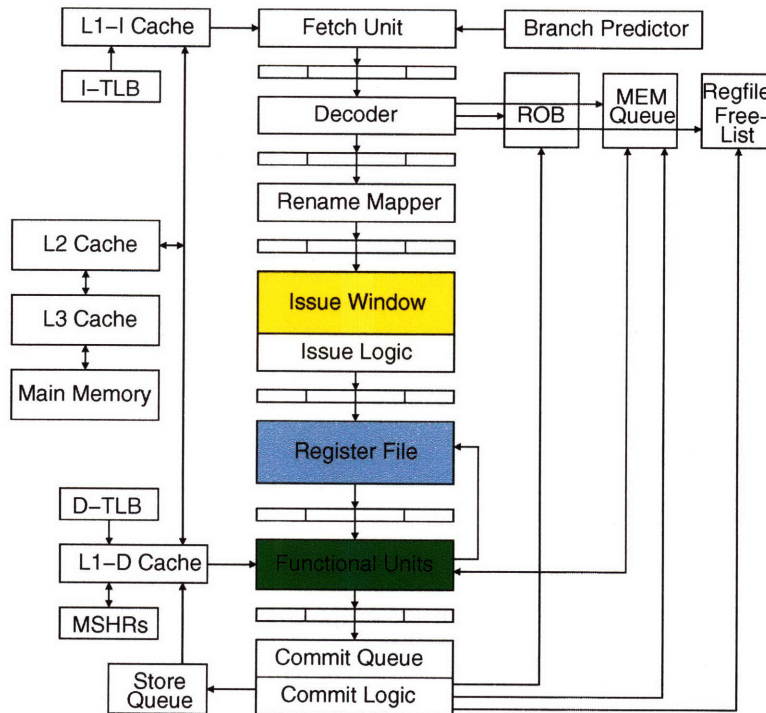
Figure 3-1: Simulation framework.

21264 [PKE$^+$03] processors, with a unified physical register file containing both speculative and committed states. Figure 3-1 illustrates the simulation framework. Instructions are fetched, decoded, renamed, dispatched, and committed in program order while being dynamically scheduled for execution.

3.1.1 Front-End Pipeline Stages

Instructions are *fetched* and *decoded* in program order. In the fetch pipeline stage, a simple global branch predictor, *gshare*, is used to speculatively fetch instructions. If a mispredicted branch instruction is detected during the execution stage, the processor flushes the pipeline, and restarts fetch along the correct path. The simulated results show that around 90% of branches are predicted correctly with a gshare predictor of 4K entries, 2-bit counters, 12-bit global history, and a 256-entry Branch Target Buffer (BTB). The processor uses the address of the first instruction in the fetch block to check if there is a cache miss. The fetch is stalled for a number of cycles according to the miss type. The processor resumes fetching one cycle after the cause of the stall has been resolved. During decode, each instruction attempts to allocate resources including: an entry in the reorder buffer to support in-order commit; a free physical register to hold the instruction's result value, if any; an entry in the issue window; and an entry in the memory queue, if this

42

is a memory instruction. A register freelist is maintained in a Last-In-First-Out (LIFO) fashion to keep track of available physical registers. If any required resource is not available, decode stalls. Otherwise, the architectural register operands of the instruction are *renamed* to point to physical registers, and the source operands are checked to see if they are already available or if the instruction must wait for the operands in the issue window. The instruction is then *dispatched* to the issue window, with a tag for each source operand holding its physical register number and readiness state.

3.1.2 Dynamic Instruction Scheduling and Execution

Our out-of-order superscalar machine issues instructions dynamically, potentially out of program order, depending on the presence of hazards. As earlier instructions execute, they broadcast their result tag across the issue window to *wake up* instructions with matching source tags. An instruction becomes a candidate for execution when all of its source operands are ready. A *select* circuit picks some subset of the ready instructions for execution on the available functional units. Priority is given to older instructions. Once the instructions have been selected for *issue*, they read operands from the physical register file and/or the bypass network and proceed to *execute* on the functional units. When the instructions *complete* execution, they write values to the physical register file and write exception status to the reorder buffer entry. When the machine knows an instruction will be executed successfully, its issue window entry can be freed.

3.1.3 Commit Stage

To preserve precise architectural state for exception handling and the illusion of sequential program execution, instructions are *committed* from the reorder buffer (ROB) in program order. If the next instruction to commit recorded an exception in the ROB, the machine pipeline is flushed and execution continues at the exception handler. As instructions commit, they free any remaining machine resources (physical register, reorder buffer entry, memory queue entry) for use by new instructions entering decode.

3.1.4 Memory Instruction Modeling

Memory instructions require several additional steps in execution. During decode, an entry is allocated in the memory queue in program order. Memory instructions are split into address calculation and data movement sub-instructions. Store address and store data sub-instructions issue independently from the window, writing

43

to the memory queue on completion. Store instructions only update the cache when they commit, using address and data values from the memory queue. Load instructions are handled as a single load address calculation in the issue window. On issue, load instructions calculate the effective address then check for address dependencies on earlier store instructions buffered in the memory queue. Depending on the memory speculation policy, the load instruction will attempt to proceed using speculative data obtained either from the cache or from earlier store instructions in the memory queue (the load instruction may later require re-execution if an address is mispredicted or a violation of memory consistency is detected). For simplicity, the simulator used in this study has a perfect memory dependency predictor (i.e. load instructions are only issued after all the store instructions that it depends on are completed). Schemes such as Store-Sets [CE98] can approximate this perfect memory dependency predictor. If the load cannot proceed due to an unresolvable address or data dependency, it waits in the memory queue to reissue when the dependency is resolved. Load instructions reissuing from the memory queue are given priority for data access over newly issued loads entering the memory queue.

The scheduler always predict that load instructions will be L1 cache hits. If the load happens to be a miss, the instructions that follow are flushed out of the execution pipeline for rescheduling and a one-cycle penalty is added to recover the pipeline. To keep track of outstanding misses, the simulator models Miss Status Handling Registers (MSHRs) [Kro81].

3.1.5 Additional Modifications

Several additional modifications are made to the original simulator to keep track of a unified physical register file organized into banks. First, a register renaming table that maps architectural registers to physical registers is added to monitor the regfile access from each instruction. To keep track of available registers and to study the performance effect of different register renaming policies on the banked regfile, a variety of register freelists are created. Extra arbitration logic is added to each regfile bank to prevent over-subscription of read and write ports when a lesser-ported storage cell is used. Since load misses are timing critical, a write-port reservation queue is also added to give them priority over other instructions. To evaluate the performance impact of banking other parts of processor design, such as organizing an N-way superscalar into N columns, changes are made to the register renaming policy, dispatch logic, wakeup-select loop, and issue logic.

Benchmark (label)	Language	Category
256.bzip2 (bzip)	C	Compression
186.crafty (crafty)	C	Game Playing: Chess
252.eon (eon)	C++	Computer Visualization
254.gap (gap)	C	Group Theory, Interpreter
176.gcc (gcc)	C	C Programming Language Compiler
164.gzip (gzip)	C	Compression
181.mcf (mcf)	C	Combinatorial Optimization
197.parser (parser)	C	Word Processing
253.perlbmk (perl)	C	PERL Programming Language
300.twolf (twolf)	C	Place and Route Simulator
255.vortex (vortex)	C	Object-oriented Database
175.vpr (vpr)	C	FPGA Circuit Placement and Routing

Table 3.1: SPEC CINT2000 benchmarks description.

3.2 Benchmarks

This thesis focuses on evaluating the performance effects of banked microarchitectures within integer pipelines, and so the SPEC CINT2000 [Hen00] benchmark suite was chosen for its wide range of applications taken from a variety of workloads. This application-based suite measures compute-intensive integer performance and comprises twelve benchmarks written in C and C++. The applications range from data compression, word processing, game playing, compiler, program language, to databases, as shown in Table 3.1. The suite has long run times but expected performance can be well characterized without running to completion. To reduce the simulation run time to a reasonable length, the methodology described in [SC00] is used to fast-forward execution to a sample of half a billion instructions for each application. The benchmarks are compiled with optimization for the Alpha instruction set.

3.2.1 Dynamic Instruction Profiling

The instruction distribution of committed instructions for each benchmark is shown in Table 3.2. The computer visualization benchmark (*eon*) is the only integer program that still contains significant floating-point instructions and has a very high proportion of load and store instructions. Applications for data compression (*bzip* and *zip*) and interpretation (*gap*) have a large fraction of arithmetic operations. It is also observed that "BRANCH" and "JUMP" form around 10%-20% of total instructions for most of the benchmarks. The combinatorial optimization program (*mcf*) is an outlier because 28% of its executed instructions are

Benchmark	FP	LOAD	STORE	JUMP/BRANCH	OTHER
bzip	0%	21%	7%	10%	62%
crafty	0%	26%	7%	13%	54%
eon	14%	25%	27%	11%	23%
gap	0%	15%	5%	4%	77%
gcc	0%	25%	13%	18%	45%
gzip	0%	20%	8%	9%	62%
mcf	0%	31%	8%	28%	34%
parser	0%	24%	8%	17%	52%
perlbmk	0%	29%	15%	15%	41%
twolf	0%	23%	6%	13%	58%
vortex	0%	25%	17%	16%	42%
vpr	0%	25%	12%	11%	52%
average	1%	24%	11%	14%	50%

Table 3.2: The instruction distribution of SPEC CINT2000 benchmarks.

control-flow instructions while the group theory interpreter (*gap*) contains only 4% of such instructions.

3.3 Baseline Superscalar Processor

Table 3.3 shows common parameters across the machines compared in this thesis. We used a large reorder buffer of 256 entries, a large memory queue of 64 entries, and 32 MSHRs such that performance is not limited by these structures. The wider fetch width (eight instructions instead of four) is used to increase the number of instructions that can be fetched per cycle with minimal hardware overhead. Since only a difference of less than 20% in IPC is observed between four-issue and eight-issue width, the improvement is not substantial enough to justify having more than a four-issue width on these codes even with the optimistic memory system. The simulation results indicate that the optimal configuration for the baseline processor is a register file of 80 entries and an issue window of 32 entries. Halving the issue window size would decrease the average IPC number by 9% but doubling the number to 64 entries would only improve the IPC by 2%.

Figure 3-2 shows the IPCs of the baseline configuration for the SPEC CINT2000 benchmark suite. The numbers range from 0.27 to 2.27 with an average of 1.51. Benchmark *mcf* has a high L1 data cache miss rate of 33% and the lowest IPC count among all the applications.

46

Item	Configuration
L1 I-cache	16KB 4-way, 64-byte lines, 1 cycle
L1 D-cache	16KB 4-way, 64-byte lines, 1 cycle
L2 unified cache	1MB 4-way, 64-byte lines, 12 cycles
L3 unified cache	8MB 8-way, 64-byte lines, 25 cycles
Fetch width	8
Dispatch, issue, and commit width	4
Integer ALUs	4
Memory instructions	2 (1-Load and 1-Store)
Reorder Buffer	256 entries
Memory Queue	64 entries
MSHR	32 entries
Branch predictor	gshare 4K 2-bit counters, 12-bit history

Table 3.3: Common simulation parameters.
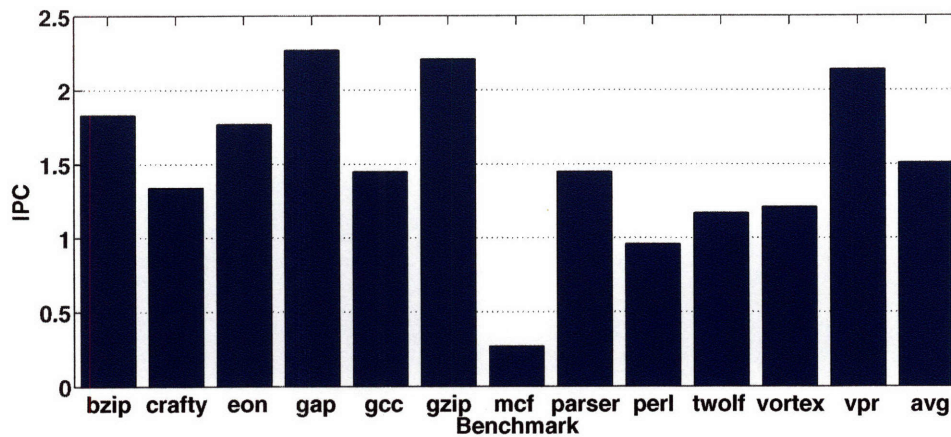
Figure 3-2: IPCs for the baseline configuration.

3.4   IPC Versus Performance

Instruction-committed-per-cycle (IPC) and clock frequency are the two most important measurements for processor performance. IPC is average the number of committed instructions for each clock cycle and it can be determined by processor pipeline simulation. The processor clock rate is governed by the circuit-level critical path, which is influenced by the entire design. Shortening the propagation delay of one component does not always translate to a faster processor. Consequently, the new clock frequency cannot be easily estimated when alterations are made to a machine. The banked microarchitectures that are proposed in this thesis typically have far less circuit latency than the traditional monolithic designs and potentially improve

processor frequency. But, lacking an actual hardware implementation, we evaluate the performance of a processor with banked microarchitectures conservatively against the idealized superscalar processor by comparing the IPCs assuming the same clock rate.

Chapter 4

A Speculative Control Scheme for Banked

Register File

Multiported register files and bypass networks lie at the heart of a superscalar microprocessor core and provide buffered communication of register values between producer and consumer instructions. Demands on both the number of ports and the number of registers in a register file grow with increasing issue widths. The trend toward simultaneous multithreading (SMT) further increases register count as separate architectural registers are needed for each thread. A banked multiported register file design can meet these increased requirements without consuming excessive power and die area. A banked regfile is a physical register file divided into multiple interleaved banks with fewer ports per bank. Bank conflicts occur when the number of simultaneous accesses to any bank is more than the number of ports on each bank. Previous proposals [WB96, BDA01, PPV02, KM03] had complex control structures to manage bank conflicts that would likely limit cycle time and add to design complexity. This chapter evaluates banked register files and proposes a simple speculative control scheme suitable for a deeply pipelined high-frequency dynamically scheduled processor. [1]

This speculative control scheme does not place any register bank arbitration in the critical wakeup-select loop, but instead speculatively issues potentially conflicting instructions. If any conflicts are found after issue, a pipelined recovery scheme quickly repairs the issue window and reissues conflicting instructions. In contrast to previous work [WB96, BDA01, PPV02], all conflicts are detected and resolved in one pipeline

---

[1]The work in this chapter was previously published in [TA03, TA05]. The 1% variation in performance results from the earlier publication is due to the fact that we use a different simulator in this thesis.

stage so that no write buffering or pipeline stalls are required. The main drawback of our scheme is the performance impact of bank conflicts and the extra pipeline stage used for port arbitration. The additional pipeline stage causes an increase in branch misprediction latency while bank conflicts add penalty cycles to repair the pipeline and delay the issue of dependent instructions.

To achieve maximum performance with the minimum number of ports per regfile bank, the number of bank conflicts is reduced by removing the correlation of accesses to the same bank and by avoiding unnecessary port contention. It is observed that instructions which become ready in the same cycle tend to be issued together and that some architectural registers are used more frequently than others. Hence, two optimization techniques, *bypass-skip* [Rus78, BDA01, PPV02] and *read-sharing* [BDA01], are investigated. Bypass-skip avoids competing for register read ports when operands will be sourced from the bypass network. Read-sharing fetches only one copy per register value from the regfile and the fetched value is shared among the instructions that requested it.

Section 4.1 describes the structure of banked multiported register files. Then, the physical characteristics of various regfile designs are evaluated and presented in Section 4.2. The simple speculative control scheme is proposed and its pipeline structure is presented in Section 4.3. In Section 4.4, a mathematical model is built to determine the port contention rate of various banked regfile configurations. Lastly, the simulated performance results are analyzed and summarized respectively in Section 4.5 and Section 4.6.

4.1   Register Bank Structure

A banked register file consists of multiple interleaved banks of non-fully-ported register cells. Figure 4-1 shows one example of a register banking scheme for a four-issue processor. The regfile provides a total of eight global read ports and four global write ports using four interleaved register banks, each with two local read ports and two local write ports. Compared to a conventional multiported structure, each word of register storage has fewer ports and the storage cell size is dramatically smaller. But now additional multiplexing circuitry is required to connect the local port bitlines to the global port bitlines, and the possibility of bank conflicts arises when too many global ports attempt to read or write the same bank.

As shown in Figure 4-1, each functional unit needs two global read ports, which are termed the left port and right port, to execute instructions with two register source operands. The local-global port crossbar is simplified by connecting one local port on each bank to only the global left operand buses, and the other

50
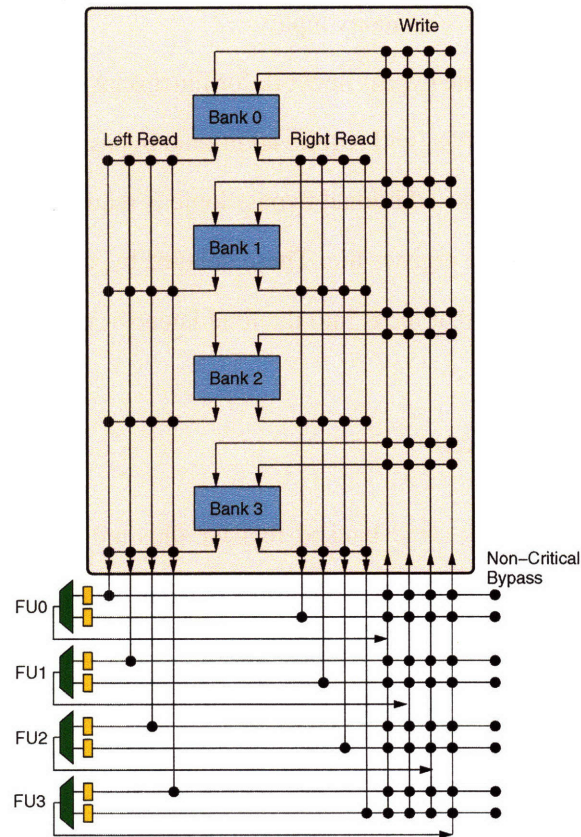
Figure 4-1: An eight-read, four-write port register file implemented using four two-read, two-write port banks. The register file interconnect and bypass network are shown as distributed muxes where each dotted crosspoint represents a potential switched connection.

local port to only the global right operand buses. Therefore, all banks have at least two local read ports. This enables any instruction to retrieve both operands from the same bank in one cycle, but doesn't allow the use of the local left ports to fetch global right port operands. Apart from the reduction in mux circuitry, this restriction simplifies port arbitration logic by cutting in half the number of possible contenders for a local read port.

In contrast, the design presented in [BDA01] employed banks with only a single read port. The single read port must connect to all global ports, and hence requires the same local-global crossbar complexity as a dual read-port design that connects each local port to half the global ports. Our initial circuit layout study indicates minimal area savings for the single read port design versus the split dual port design once the cost of the local-global crossbar is included. Moreover, the single read port bank requires considerably more complicated control logic to handle execution of an instruction that fetches both operands from the same

51

bank, and each read port arbiter has twice as many inputs.

Figure 4-1 also shows a portion of the bypass network for functional units with single cycle latency; critical multiple cycle units, such as load units, will require additional bypass paths. Register file writeback may require one or more cycles, in which case additional bypass logic is required for results that have completed but which are not yet available from the register file. These delayed bypass paths are not latency critical and can be supplied by an early stage mux that feeds into the final latency-critical mux stage [FGK+02].

4.2 Physical Characteristics

The physical characteristics of various sized banked register files are investigated and presented in this section. All designs were laid out in a 0.25 μm CMOS process from TSMC. The storage cells are a standard six transistor SRAM design, with differential write ports and single-ended read ports. Section 4.2.1 describes the layouts of various designs that have been undertaken to determine area, access latency, and energy consumption tradeoffs of regfile designs. First, the die area is estimated by measuring the actual layout in Section 4.2.2. Regfile delay and energy numbers were obtained from HSPICE [Nag75] simulations of extracted layouts with a 2.5 V supply voltage in Section 4.2.3.

4.2.1 Regfile Layout

Metal 1 is used for local bitlines within a bank and metal 2 for word lines. The local ports from each bank then connect to the global bitlines running over the cells in metal 3. Most previous work has assumed that a large conventional multiported register file would have each port on a storage cell connected directly to the global bitline. With more metal layers, it is desirable to employ a hierarchical bitline structure, where each port on a cell connects to a local bitline which in turn connects to the global bitline [AKSB01, FGK+02]. On each access, only one local bitline is connected to the global bitline. The parasitic drain capacitances of the storage cells in other banks are not driven, reducing delay and energy dissipation. Another benefit is that signal-to-noise ratio improves in the presence of leakage currents from off cells [AKSB01]. Adopting hierarchical bitlines in our baseline flat design reduces the relative energy and delay advantages of a multibanked design. To save area, we employ a single-ended global write bitline which is converted to a differential local bitline using a local inverter. To further save area, we pack two local storage cells into one global bit column where possible. This has three disadvantages: a 2:1 column mux is required which adds
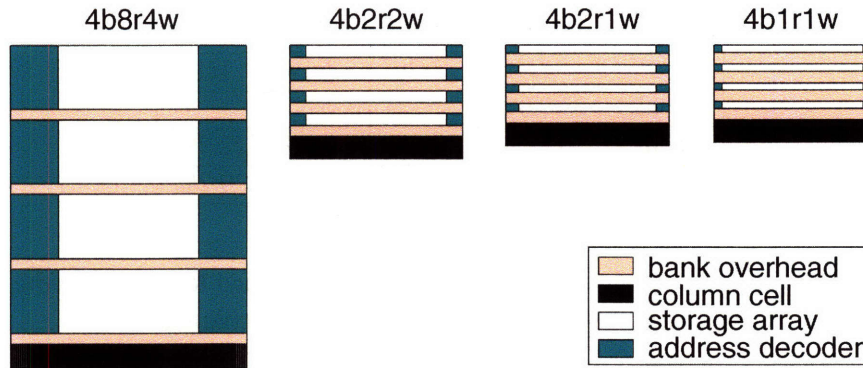
Figure 4-2: Area comparison of four different 64×32b regfiles for a quad-issue processor. The clear regions represent the storage cells while the lighter shaded regions represent the overhead circuitry in each bank. The black shading at the bottom is the area required for the global bitline column circuitry. The medium-dark shading to the side is the area for address decoders.

area and delay; greater wordline load also adds to delay; and twice as many local bitlines are discharged on each access, which increases energy usage.

Figure 4-2 provides a graphical comparison of the floorplan of four 64×32-bit 8 read-port and 4 write-port multibanked register file designs. Each configuration is labeled as (*#banks*)b(*#reads*)r(*#writes*)w, where (*#banks*) is the number of banks, (*#reads*) is the number of local read ports, and (*#writes*) is the number of local write ports. For example, *4b8r4w* refers to a regfile with four interleaved banks of fully ported cells. As found in the initial study, multiplexing circuits dominate the area of small ported multibanked designs. Moving from a single read port to split dual read ports per bank has minimal impact on the area. In addition, the banked design with only a single read port cannot sustain eight global read port accesses and relies on the bypass network to supply the missing read operands.

4.2.2   Regfile Area Comparison

The relative area of a variety of 64×32-bit 8 read-port and 4 write-port multibanked register file designs are shown in Table 4.1, and the detailed area breakdown studies are shown in Figure 4-3. For the designs with 8 read ports and 4 write ports per storage cell, moving to hierarchical bitlines adds area because of the interconnection overhead. An additional 11% area overhead is incurred when there are 16 words per local bitline, and an additional 23% moving to 8 words per local bitline. These designs do not have bank conflicts.

As the number of local ports per bank is reduced, area drops dramatically. Compared to the fully ported monolithic regfile, designs with four banks of less ported cells are around one quarter the size, and designs

64×32b, 8 read ports, 4 write ports				
Area	8r4w	2r2w	2r1w	1r1w
1 banks	100.0%	-	-	-
4 banks	111.0%	29.0%	24.3%	22.9%
8 banks	122.6%	37.1%	32.0%	30.2%
Packing	1	2	2	2

Table 4.1: Relative area of different 64×32-bit eight global read port and four global write port register file designs. Packing is the number of local bit cells packed per global bit column.
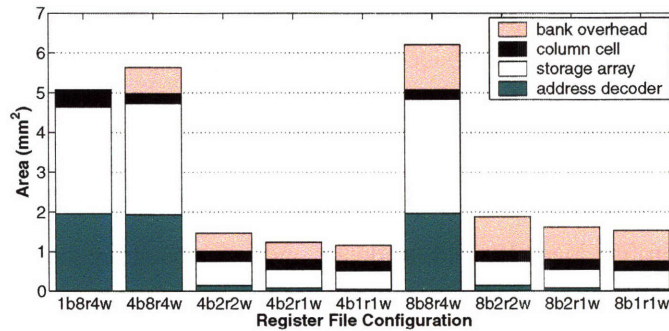
Figure 4-3: Detail area breakdown of various 64×32b eight read-port and four write-port register file designs.

with eight banks are around one third the size. Apart from the reduction in storage cell size, regfiles with a smaller numbers of ports per bank have significantly less address decoder area than the highly multiported one. Each bank has fewer decoders with narrower addresses. Multiplexing overhead dominates when there are only a few ports per cell, as seen in both Figure 4-2 and Figure 4-3. Designs with two read ports per bank are only a few percent larger than designs with a single read port per bank given that the single read port must connect to all global read ports whereas each of the two read ports only connects to half of the global read ports. Also, increasing the number of write ports from one to two adds only 16–20% in area.

4.2.3 Regfile Delay and Energy Evaluation

Table 4.2 lists normalized delay and energy measures for various multibanked register file designs in comparison to the unified design. Figure 4-4 shows the detailed delay and energy breakdowns of these designs. For regfiles with fully ported storage cells, using hierarchical bitlines reduces energy by almost 40% and cuts delay by 8–17%. The latency is further decreased, up to around 25%, when designing with lesser-ported cells. A slight decline in energy is also noticed for these lesser-ported banked regfiles. The delay and energy

savings, however, are not as great as might be expected from the large area reduction. The packing of two local storage cells per global bit column slows the wordline drive and adds a column mux stage, and it also causes twice as many bitlines to discharge on a read. Alternatively, the less ported cells can be reoptimized for even smaller delay and energy, but this would add considerable additional area.

64×32b, 8 read ports, 4 write ports				
Delay	8r4w	2r2w	2r1w	1r1w
1 bank	100.0%	-	-	-
4 bank	92.4%	79.1%	79.0%	81.9%
8 bank	83.3%	74.8%	74.8%	77.1%
Energy	8r4w	2r2w	2r1w	1r1w
1 bank	100.0%	-	-	-
4 bank	62.0%	57.9%	56.9%	40.5%
8 bank	61.4%	58.6%	57.6%	40.7%
Leakage	8r4w	2r2w	2r1w	1r1w
SRAM	100.0%	40.4%	29.3%	25.3%

Table 4.2: Relative delay, energy, and leakage numbers of different 64×32-bit eight global read port and four global write port register file designs.

The primary source of energy dissipation for the 0.25 μm CMOS process is dynamic switching of load capacitance. Within a few process generations, it is expected that static leakage current will be responsible for a large fraction of total power dissipation [CBF00]. Table 4.2 shows the relative size of leakage energy across the three designs. The relative leakage energy numbers were obtained by calculating the total width of leaking transistors, assuming that 80% of stored values are zero, and that the bit cell ports were optimized to reduce read energy for zero values [TA00]. Banking reduces leakage power by at least 60% over the baseline case, and so we expect even greater relative power savings as leakage currents grow.

4.3 Control Logic

A banked multiported register file can provide sufficient bandwidth for a superscalar processor at a lower cost than a flat design. The main challenge is devising control logic that can handle the inevitable bank conflicts without compromising cycle time or adding excessive complexity.
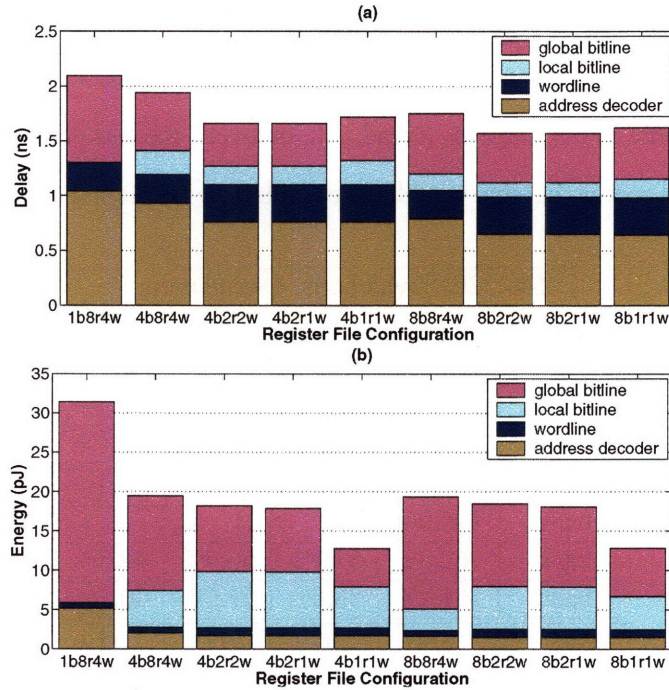
Figure 4-4: Detail breakdown of various $64 \times 32b$ eight read-port and four write-port register file designs in terms of (a) read access delay and (b) read energy consumption.

4.3.1 Speculative Pipeline Control Scheme

Previous work has either placed additional arbitration logic in the select path to avoid conflicts or required that multiple pipeline stages be stalled [WB96, BDA01, PPV02, KM03]. Both approaches complicate critical timing loops [BTME02]. In particular, stalling a deep pipeline is usually prohibited in high-frequency processors due to the difficulty of generating and routing a global stall signal to a large number of pipeline registers. To overcome these problems, we introduce a speculative pipeline control scheme. The pipeline first speculatively issues potentially conflicting instructions then performs port arbitration in a later pipeline stage. If any conflicts are detected after issue, a pipelined recovery scheme, described below, quickly repairs the issue window and reissues conflicting instructions. Figure 4-5 compares this modified processor pipeline to the baseline design.

A conventional pipeline has a fixed mapping of issued instruction operands to register file ports, so operands of an instruction can be fetched from the regfile immediately after issue. A banked regfile scheme, however, must first mux operand addresses into the available register file ports. The extra arbitration pipeline stage shown in Figure 4-5 detects both read and write regfile conflicts and muxes the winning addresses into

56

Fetch	Decode	Rename	Issue _{Wakeup Select}	Read Bypass	Execute	Writeback	Commit		Baseline

Fetch	Decode	Rename	Issue _{Wakeup Select}	Arbitrate	Read Bypass	Execute	Writeback	Commit	Modified
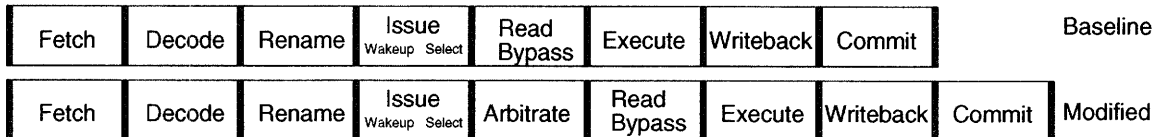
Figure 4-5: Pipeline structures of processor with unified register file and processor with multibanked register file. An additional cycle is added for multibanked register file for read port arbitration and muxing. Read bank and write bank conflicts are also detected in this cycle.

the address decoders. The arbitration stage also manages requests to write back results from long or variable latency operations such as divides or cache misses. These are given higher priority than newly issued instructions. In Figure 4-5 and in the evaluation model, a whole pipeline stage is allocated to the arbitration and register address mux, but the actual penalty should be much lower in practice.

In the modified pipeline, all instructions that pass the arbitration phase can read from and write back to the regfile with no conflicts. The speculative control scheme avoids register bank write buffers [BDA01], which increase the size of the bypass network and require pipeline stalls when full. It should be noted that sufficient write ports must be provided such that write bank conflicts do not cause a large performance degradation. This approach is also much simpler than schemes that delay physical register allocation until writeback to avoid conflicts [PPV02].

4.3.2   Repairing the Issue Window

The arbitration stage detects all bank conflicts, that is, when too many reads try to access the left or right side of a single bank or too many writes try to access the same bank. If such conflicts are detected, the processor must repair the issue window and reissue the conflicting instructions. Figure 4-6 shows the method by which the pipeline state is restored after a conflict. A second group of instructions following the ones that encounter a conflict will have been speculatively issued into the pipeline in parallel with the detection of the conflict. This second group is killed along with the instructions in the first group that were not granted a read or write port. The wakeup phase that would have been used to broadcast the tags of the second group of instructions is now used to repair the issue window by broadcasting the tags and resetting the ready and issued bits for the destinations of the killed instructions in the first group. Issue will now resume correctly in the select phase of this pipeline stage. This approach adds only a mux into the tag broadcast path of the critical wakeup phase.
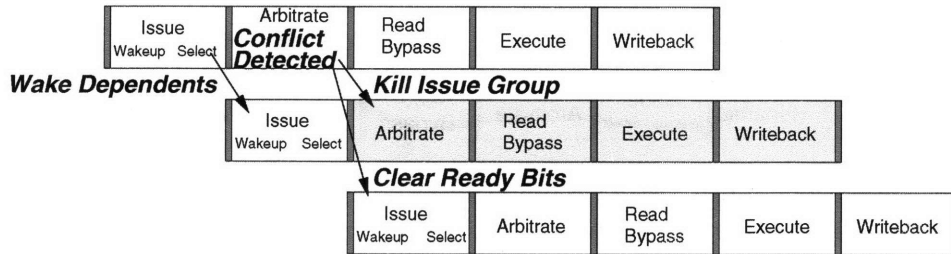
Figure 4-6: Pipeline diagram shows repair operation after conflicts are detected. The wakeup tag search path is used to clear ready bits of instructions that had a conflict causing them to be reissued two cycles later. Any intervening instruction issues are killed.

4.3.3 Conservative Bypass-Skip

It has been previously noted [TA00, BDA01, PPV02], that a great percentage of operands are either the dedicated zero register (R0 on MIPS, R31 on Alpha) or are supplied from the bypass network. The number of requisite read ports can be reduced significantly if we have a separate zero input to the bypass mux, and also if operands that will be sourced from the bypass network do not compete for access to the register file ports.

Avoiding read port contention for bypassed operands would at first appear to require that the arbitration logic wait until the bypass logic determines if operands will be bypassed. To avoid this increase in pipeline latency, the check can be folded into the wakeup phase. Park et al. proposed an optimistic bypass hint scheme [PPV02] where an extra hint bit is added to each operand of instructions waiting in the issue window. The hint bit is cleared if the operand was ready before the instruction entered the issue window, otherwise it is set. When the instruction is selected, an operand with the hint bit set will not contend for a read port as it is likely to be sourced from the bypass network. The disadvantage of this scheme is that it is only a prediction, which when incorrect requires stalling earlier pipeline stages to allow the instruction to access the read ports. Stall are difficult to implement in a high frequency pipeline without compromising cycle time. The mispredictions also reduce performance in cases where the read could have been satisfied from a free read port if only it had been in contention.

Instead, we adopt a conservative bypass bit scheme, which is always correct but which only avoids contending for read ports for values bypassed from the immediately preceding cycle. Each operand in the instruction window has multiple comparators, one per tag-wakeup port, to determine if the operand becomes ready in that cycle. The bypass bit for each instruction window operand is stored in a latch that is loaded
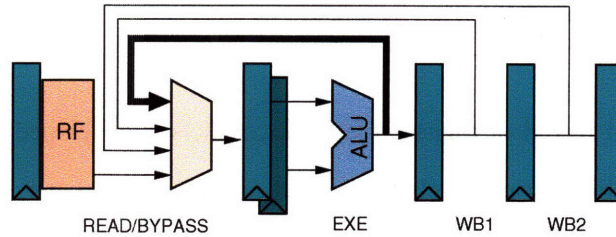
Figure 4-7: Conservative bypass skip only avoids read port contentions when the value is bypassed from the immediately preceding cycle.

with the result of OR-ing together comparator results every cycle. If an instruction is selected in the same cycle where a tag match caused the instruction to wake up, the bypass bit will be set indicating that the value is available from the bypass network. If the instruction is not selected for issue, the bypass bit latch will be cleared by the failing tag matches on the next wakeup phase. The bypass bit is conservative because it is only set for values that will be ready in the cycle before the current instruction executes as shown in Figure 4-7. Where there are several pipeline stages feeding the bypass mux (e.g., when register file access takes multiple cycles), this scheme will still compete for read ports even though these operands will be sourced from later pipeline stage bypasses. In practice, our simulation results show that this lost opportunity causes negligible performance impact. To reduce datapath complexity, a microarchitecture might not support bypass from every functional unit. In this case, the wakeup tag search can be modified to broadcast a signal indicating whether the operand can be bypassed. This value can then be latched into the bypass bit on a successful tag match. Any register operand of an issued instruction which doesn't have the bypass bit set must contend for read ports.

4.3.4 Read Sharing

Read bank conflicts commonly occur when multiple instructions in an issue group try to read the same physical register [BDA01]. Instructions that depend on the same register become ready on the same cycle and are likely to be issued together. The read port arbiter can detect this sharing and remove the conflict by setting enable signals such that a local port drives multiple global ports. Since the banked regfile structures presented in this chapter have limited local-to-global crossbars, read sharing is localized to the left ports and the right ports. For example, if a physical register is read from both sides, two local ports are required.

4.4 Modeling Local Port Contention

In this section, an analytical model of register file bank conflicts is developed under the assumption that register file accesses are uniformly randomly distributed across banks. Later, this model is used to help understand the results obtained by detailed microarchitectural simulation.

4.4.1 Port Conflict Probability (PCP)

Bank conflicts occur when too many instructions compete for the same bank port during the same cycle. The port conflict probability (PCP) is the probability of having any conflicts in any regfile bank at a given cycle. Equation 4.1 shows how PCP is calculated for a regfile with two interleaved banks, $Bank0$ and $Bank1$. By definition, PCP is the probability of either $Bank0$ or $Bank1$ encountering any conflict, $P(Conflict_{Bank0} \cup Conflict_{Bank1})$. This is further derived to the summation of each bank conflict probability $(P(Conflict_{Bank0}) + P(Conflict_{Bank1}))$ minus their interceptions $(P(Conflict_{Bank0} \cap Conflict_{Bank1}))$. For a regfile with fully ported storage cells, PCP is always zero.

$$
\begin{aligned}
PCP = {} & P(Conflict_{Bank0} \cup Conflict_{Bank1}) \\
= {} & P(Conflict_{Bank0}) + P(Conflict_{Bank1}) - \\
& P(Conflict_{Bank0} \cap Conflict_{Bank1})
\end{aligned}
\tag{4.1}
$$

By knowing the number of simultaneous regfile accesses and the configurations of banked regfile structure, PCP can be easily computed for each clock cycle. We first need to determine M (Equation 4.2), which is the maximum number of banks that could encounter conflicts in a cycle, given A accesses to a regfile with B banks of N ported storage cells. When M is zero, we have a conflict-free cycle ($PCP = 0$). When M is one, we have the possibility of conflicts in only a single bank (e.g., $P(Conflict_{Bank0} \cap Conflict_{Bank1})$ is zero). When M is greater than one, we could have conflicts across multiple banks in the same cycle. Depending on the value of M, PCP is calculated differently (Equation 4.3). Since there are only two possible states, *conflict* and *non-conflict*, for each regfile bank, binomial coefficients are used to find conflicting sets (Equation 4.4). $_AC_x$ determines the number of possible combinations where x out of A regfile accesses fetch values from the same bank regardless of the ordering and $\frac{(B-1)^{A-x}}{B^{A-1}}$ is its probability. $O(A, B, N)$ adjusts for duplicate counts where multiple banks encounter bank conflicts in the same cycle.

$$M = \min(B, \lfloor \frac{A}{N+1} \rfloor) \tag{4.2}$$

$$PCP = \begin{cases} 0 & if\ M < 1 \\ \displaystyle\sum_{x=N+1}^{A} {}_AC_x \cdot \frac{(B-1)^{A-x}}{B^{A-1}} & if\ M = 1 \\ \displaystyle\sum_{x=N+1}^{A} {}_AC_x \cdot \frac{(B-1)^{A-x}}{B^{A-1}} - O(A, B, N) \\ & if\ M > 1 \end{cases} \tag{4.3}$$

$$_AC_x = \frac{A!}{x!(A-x)!} \tag{4.4}$$

Analyzing the conflict overlap, $O(A, B, N)$, between multiple banks, two sets of binomial coefficients are required to find the two orthogonal sets of combination. One is for the number of banks while the other is for the number of accesses. In Equation 4.5, $_BC_d \cdot F(d, A, B, N)$ determines the number of instances where at least d out of B banks encounter conflicts and $\frac{1}{B^A}$ is its event probability. Table 4.3 shows how $F(d, A, B, N)$ can be found for different numbers of overlapping conflict banks.

$$O(A, B, N) = \sum_{d=2}^{M} (-1)^d \cdot \frac{1}{B^A} \cdot {}_BC_d \cdot F(d, A, B, N) \tag{4.5}$$

d	$F(d, A, B, N)$
2	$\sum_{j_0=N+1}^{A-(N+1)} \sum_{j_1=N+1}^{A-j_0} {}_AC_{j_0} \cdot {}_{A-j_0}C_{j_1} \cdot (B-2)^{A-j_0-j_1}$
3	$\sum_{j_0=N+1}^{A-2(N+1)} \sum_{j_1=N+1}^{A-j_0-(N+1)} \sum_{j_2=N+1}^{A-j_0-j_1} {}_AC_{j_0} \cdot {}_{A-j_0}C_{j_1} \cdot {}_{A-j_0-j_1}C_{j_2} \cdot (B-3)^{A-j_0-j_1-j_2}$
4	$\cdot$
$\cdot$	$\cdot$
$\cdot$	$\cdot$

Table 4.3: $F(d, A, B, N)$ for various d values.

4.4.2 PCP Analysis

The performance sensitivity of different banked regfile configurations can be assessed by PCP if its accesses are not correlated. Figure 4-8(a) plots the PCPs of a regfile with sixteen banks of varying number of local
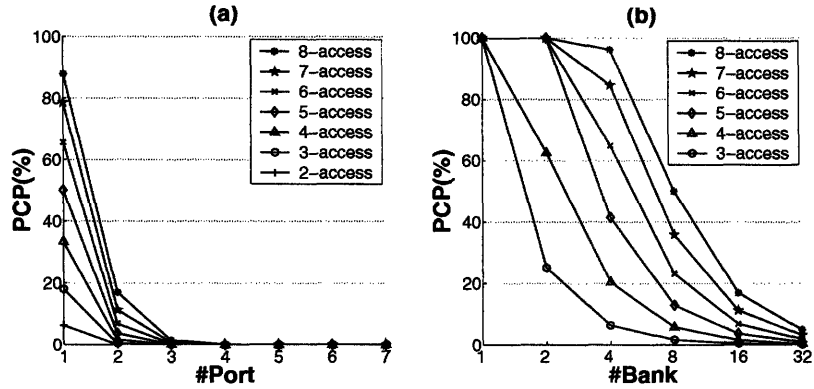
Figure 4-8: PCP for designs with (a) 16 banks with varying number of local ports and (b) varying number of banks with two local ports.

ports over a number of accesses. It shows that having the additional second local port reduces the number of conflicts by at least a factor of four, even for four-issue machines. Similarly, Figure 4-8(b) plots the PCPs of a regfile with varying number of banks of two local port cells over a range of accesses. This demonstrates that PCP is very sensitive to the number of banks when storage cells are lesser-ported, and having a sufficient number of banks is crucial to keep the conflicts low. For example, PCP is limited to around 5% when there are four accesses to a eight-banked structure, but it grows exponentially to 50% after doubling the number of accesses.

4.5 Simulation Results

To evaluate the performance impact of the proposed banked regfile, the simulator is modified to keep track of a unified physical register file organized into banks. The branch misprediction latency is increased by one cycle to account for the extra arbitration cycle in the speculative control scheme. Controls for pipeline repairs are also modeled to include the penalties when bank conflicts occur.

4.5.1 Performance Sensitivity

Figure 4-9 and Table 4.4 show the resulting absolute and relative IPC numbers obtained for a four-issue machine with an 80-element register file. The baseline processor is simulated on the standard eight-stage pipeline with a fully ported regfile, while others are simulated on the modified nine-stage pipeline with a banked regfile. Each configuration is labelled as (#banks)B(#reads)R(#writes)W(bypass?)(sharing?), where

62
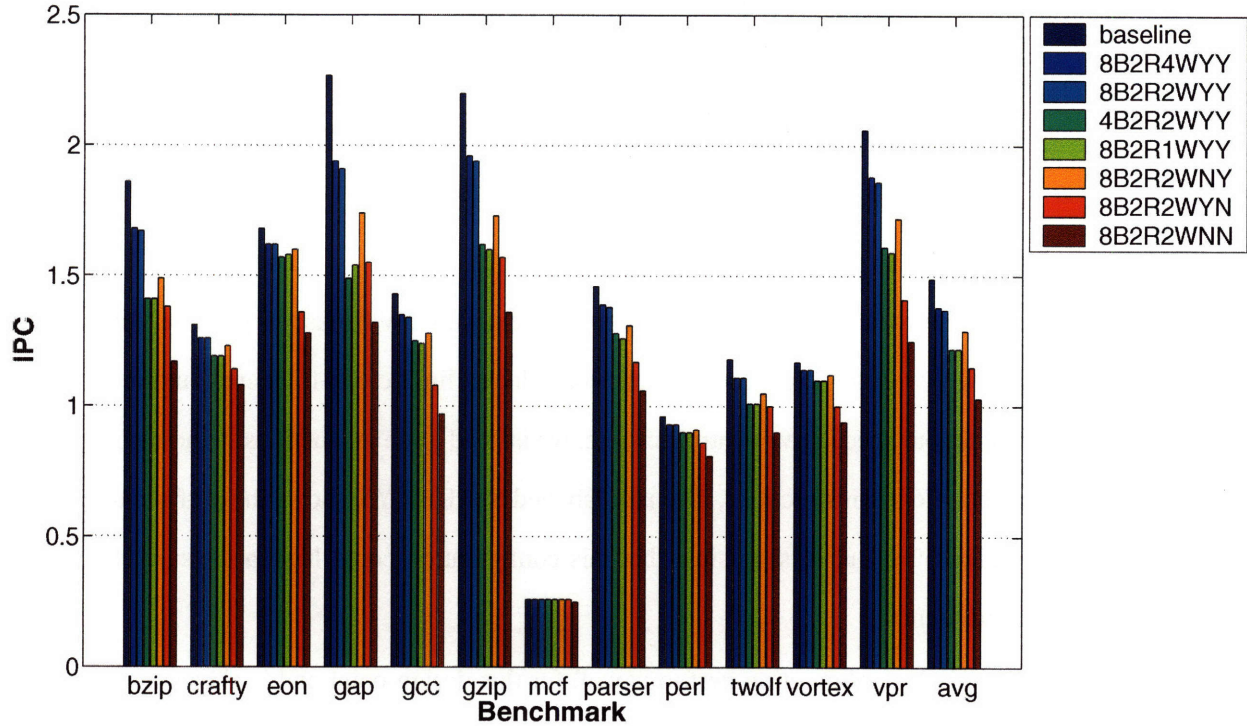
Figure 4-9: IPCs for the 4-issue pipeline with register file of size 80.

(*#banks*) is the number of banks, (*#reads*) is the number of local read ports, (*#writes*) is the number of local write ports, (*bypass?*) indicates if the processor avoid competing for register ports when the value bypassed from the last execution cycle, and (*sharing?*) indicates if local read ports can drive multiple global read ports to implement read sharing.

The simulations show that the performance degradation would be less than 1% compared to the baseline if the select logic were changed to avoid register bank conflicts at issue time, and where both bypassing

(in %)	bzip	crafty	eon	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr	avg
Baseline	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
8B2R4WYY	90.3	96.2	96.4	85.5	94.4	89.1	100.0	95.2	96.9	94.1	97.4	91.3	93.9
8B2R2WYY	89.8	96.2	96.4	84.1	93.7	88.2	100.0	94.5	96.9	94.1	97.4	90.3	93.5
4B2R2WYY	75.8	90.8	93.5	65.6	87.4	73.6	100.0	87.7	93.8	85.6	94.0	78.2	85.5
8B2R1WYY	75.8	90.8	94.0	67.8	86.7	72.7	100.0	86.3	93.8	85.6	94.0	77.2	85.4
8B2R2WNY	80.1	93.9	95.2	76.7	89.5	78.6	100.0	89.7	94.8	89.0	95.7	83.5	88.9
8B2R2WYN	74.2	87.0	81.0	68.3	75.5	71.4	100.0	80.1	89.6	84.7	85.5	68.4	80.5
8B2R2WNN	62.9	82.4	76.2	58.1	67.8	61.8	96.2	72.6	84.4	76.3	80.3	60.7	73.3

Table 4.4: Normalized IPC % for a quad-issue machine with 80 physical registers. Configurations are labelled as (#banks)B(#local read ports)R(#local write ports)W(bypass skipped?)(read sharing?). Results are normalized to the IPC of the baseline case (unified with eight read and four write ports).

Category	Workload	Average IPC
1-Thread	One thread of a SPEC CINT2000 benchmark	1.79
2-Thread	Two threads of different SPEC CINT2000 benchmarks	2.60
4-Thread	Four threads of different SPEC CINT2000 benchmarks	3.42

Table 4.5: Three workload categories.

and port sharing are used to reduce conflicts in a system with eight 2R2W banks. In practice, this scheme would have a much slower wakeup-select loop which would likely limit clock frequency and reduce total performance. By adopting our speculative control scheme, we instead issue instructions without considering conflicts and later kill conflicting instructions. The row labelled 8B2R2WYY shows the performance drops another 0–15% averaging 5%. But it is possible that this configuration could have a lower cycle time to make up for this IPC difference.

Overall, the 8B2R2WYY configuration performs well for this design point and is chosen as the center point in perturbing other parameters. Reducing the number of banks to four (4B2R2WYY), lowers performance by another 0–14%. We can also see that moving from 1 to 2 write ports (8B2R1WYY, 8B2R2WYY) improves performance by more than 9% but having more than 2 write ports per bank (8B2R4WYY) only improves performance by another 0.4%. This is expected given that average IPCs are rarely above 2, and some instructions do not write to registers.

The two optimization techniques, conservative bypass-skip and read sharing, greatly reduce the number of bank conflicts and have a significant effect across all the benchmarks. The performance degrades by over 5% when bypass optimization is removed (8B2R2WNY), around 13% when the sharing optimization is removed (8B2R2WYN), and a total of 20% when both are removed. Our simulations confirm the observation in [BDA01] that groups of load and store instructions dependent on the stack pointer tend to issue together, probably at procedure call/return points. It is also worth noting that branch instructions dependent on the same register were another common source of read sharing.

4.5.2 Extending to SMT Architecture

To evaluate the scalability of this work, the proposed banked register file scheme is extended to SMT processors. One might expect that the higher utilization of an SMT processor would raise the number of bank conflicts and hence reduce the applicability of our banked regfile design. Instead, the initial data surprisingly

Name	Applications
mix2.1	(bzip, twolf)
mix2.2	(gap, gzip)
mix2.3	(crafty, gcc)
mix2.4	(mcf, perl)
mix2.5	(parser, vortex)
mix2.6	(eon, vpr)
mix4.1	(crafty, gcc, gzip, votex)
mix4.2	(bzip, eon, mcf, twolf)
mix4.3	(gap, parser, perl, vpr)

Table 4.6: Heterogeneous multithreaded workloads.

reveals that banking produces even better results for an SMT core than a single thread superscalar core. This result is partly due to SMT's ability to hide the increased branch mispredict penalty, i.e., when one thread experiences a misprediction, other threads can continue to execute instructions. The positive results demonstrate that the speculatively controlled banked regfile works well for highly utilized SMT processors and has the potential to cope with their ever-increasing regfile port and size requirements.

Table 4.5 shows the three workload categories that were used for performance evaluation and their average IPC. The *1-thread* workload models the behavior of superscalar processors, while *2-thread* and *4-thread* workloads account for the multithreading environment of SMTs. For the multithreading workloads, we randomly paired different combinations of the benchmark as listed in Table 4.6. Changes are made to the baseline configuration to accommodate SMT's processing requirements. The issue width is increased to eight where four memory instructions are allowed (two loads and two stores), the regfile and ROB size is increased to 512 entries, and the memory queue size is increased to 128. However, the issue queue and MSHR size remains the same, 32 entries, as we found that larger sizes do not improve performance significantly.

Figure 4-10 shows the resulting absolute IPC obtained for *1-thread*, *2-thread*, and *4-thread* workloads. The *1-thread* workload models the behavior of superscalar processors, while *2-thread* and *4-thread* workloads account for the multithreading environment of SMTs. It is found that the average performance degradation of the 16B2R2WYY design increases from 5%, 8%, to 12% as we increase the workload from one, two, to four threads, respectively. This indicates that the register file does not provide sufficient throughput by having only two local read ports. After adding another pair of local read ports to the register file banks
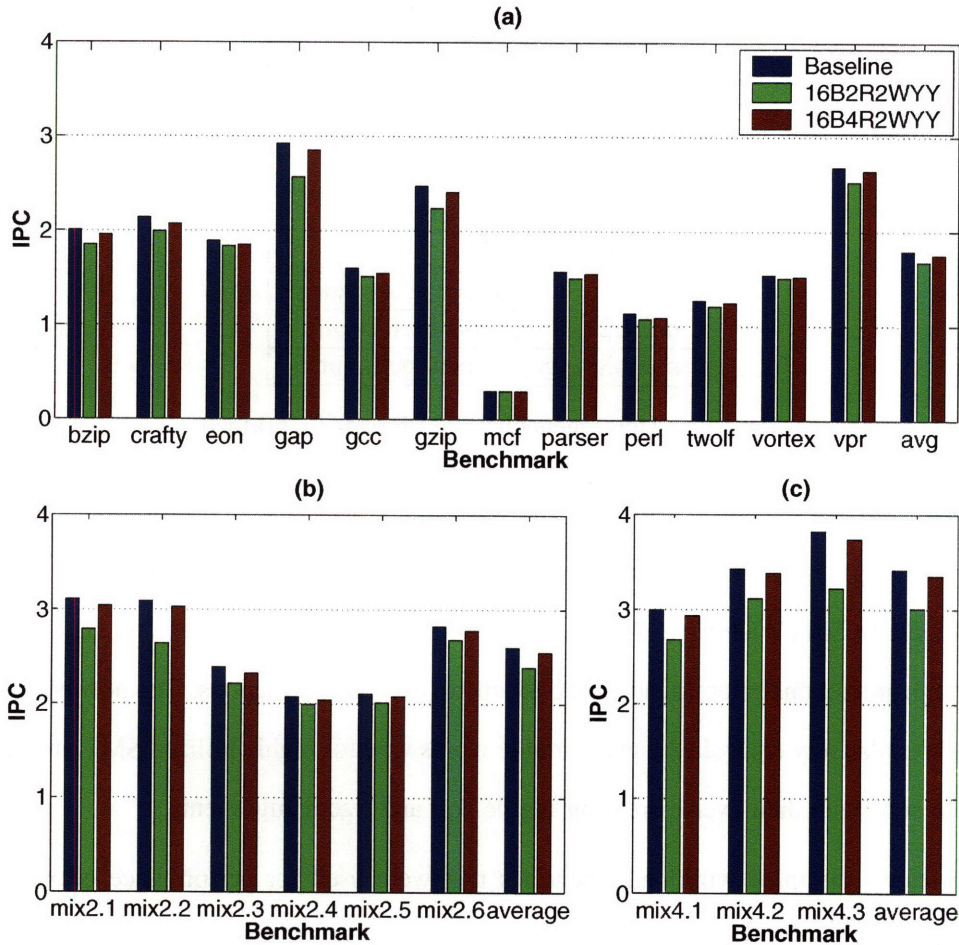
Figure 4-10: IPCs for (a) 1-Thread, (b) 2-Thread, and (c) 4-thread workloads.

(16B4R2WYY), performance is restored to just a 2% IPC degradation in all three workload categories. In fact, the average IPC increases as we increase the workload, but the percentage of IPC degradation and its variation across applications decreases slightly.

4.5.3 In-order Superscalar

A four-way in-order superscalar also needs a highly ported register file to accommodate a maximum of eight reads and four writes per cycle. Figure 4-11 shows that using an eight-banked regfile with two local read ports and two local write ports decreases IPC by less than 5% on average, with a range from 0 to 8%, in comparison to a fully ported design. This result is similar to the one that we observed with out-of-order superscalar processors. Regardless of issue policy, the proposed banked regfile performs consistently across different machine configurations and architectures.
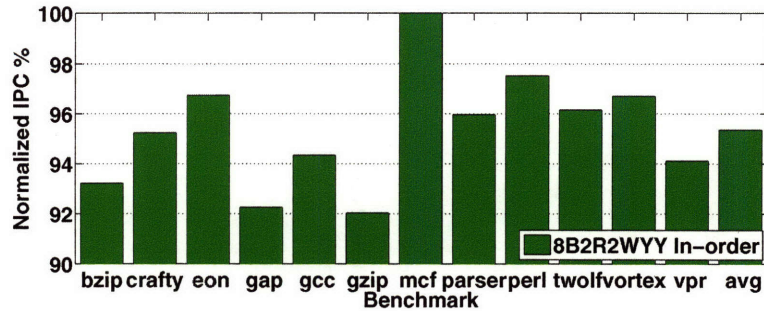
Figure 4-11: Normalized IPC % for a four-way in-order machine with a 8B2R2WYY regfile. Results are normalized to the IPC of a four-way in-order superscalar with a fully-ported regfile.

4.5.4 Correlation Among Accesses

To compare the observed conflicts against those predicted by the model developed in Section 4.4, the simulator is used to gather a histogram of the number of accesses to the regfile in each cycle. Then, the weighted sum of PCP is calculated over the whole run. These numbers are compared against simulation results for *4-thread* workloads on 16B4R2W regfiles with different additions of optimization techniques as shown in Figure 4-12. Without the read port optimizations there are many correlated read accesses leading to a much higher conflict rate than that predicted by uniform random accesses. Applying *bypass-skipped* reduces the number of correlated accesses somewhat, but applying *read-sharing* causes fewer conflicts than a random distribution. By applying both optimization techniques, a slightly lower percentage of conflicting cycles than purely random accesses is also achieved. Read-sharing is very effective in removing the correlations between read accesses. The observed difference is because if each access is truly mutually exclusive, we would still have a few situations where multiple instructions are reading the same registers and cause conflicts. These conflict events are included in the uniform random accesses assumption model, but not in the simulation with the sharing optimization. Read sharing avoids these type of bank conflicts by providing values to the same half of read ports. However, for write port conflicts, the simulation numbers indicate that the write port allocation is not uniformly distributed across all the banks. Unlike for reads, no optimization techniques are applied to remove or prevent write correlations. Several register renaming policies, including FIFO, LIFO and Random, were also investigated to determine if different algorithms would have some impact on the correlations but our results were unchanged for all cases.
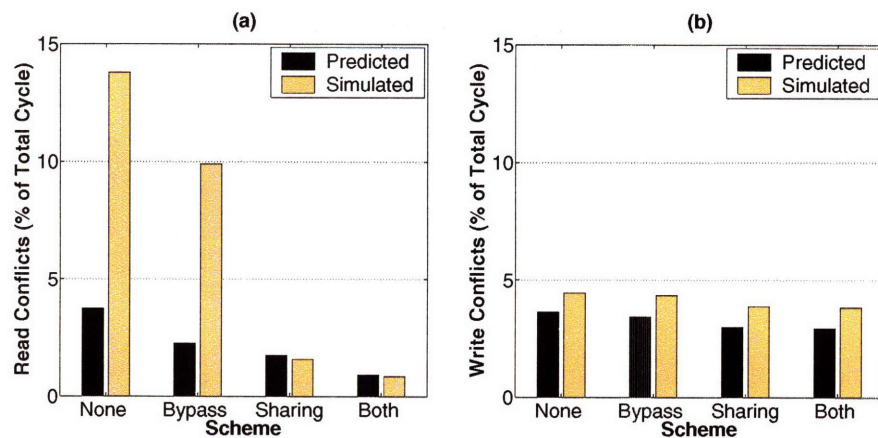
Figure 4-12: Conflict cycle comparison for (a) reads and (b) writes.

4.6 Summary

In this chapter, I have presented an energy-efficient banked multiported regfile design together with a much simpler pipeline control scheme suitable for a high-performance dynamically scheduled processor. To prevent increases in cycle time, the speculative control avoids register bank arbitration in the already timing-critical issue logic but instead adds an extra pipeline stage to detect and resolve any conflicts. Since bank conflicts can degrade performance, the number of conflicts is kept low without buffering by using a sufficient number of banks and ports and by removing the correlation between accesses to the same bank. Through layout studies, it is shown that for a small number of ports per bank, overall register file size grows slowly as ports are added because area is dominated by bank interconnect. This fact is exploited by using more ports per bank to reduce the IPC impact of the proposed speculative pipeline control scheme. The reduction in control logic complexity and bypass mux size of the speculative design can justify the slight increase of the overall regfile area over the minimally ported one.

For a four-issue superscalar processor, we reduce the 80-entry regfile size by over a factor of three, access time by 25% and access energy by 40%, while reducing IPC by 6% with the 8B2R2WYY configuration. For an eight-issue SMT processor, the area of the 512-entry regfile is reduced by a factor of seven, access time by 30%, and energy by 60%, while IPC is degradated by less than 2% with the 16B4R2WYY configuration. I expect the power savings of using smaller and less ported storage cells to increase as leakage rises with decreasing feature size. Even though this banked regfile design exhibits a small performance penalty, the reductions in register file delay can potentially be used to increase the clock rate and lead to a

68

more complexity-effective design. The ability to reduce area and power significantly with minimal performance degradation should also make this approach attractive for multi-processor chips which are designed to provide the highest possible thread throughput at low cost.

Chapter 5

RingScalar: A Complexity-Effective Banked Architecture

Banked regfile structures have been shown to be very effective in the previous chapter. In this chapter, a new centralized out-of-order superscalar architecture, RingScalar, comprised of banked microarchitectures arranged in a ring configuration is introduced. RingScalar is a complexity-effective architecture that simplifies all the major components in the instruction flow to increase area and power efficiency without excessive pipeline control complexity. It builds an N-way superscalar from N columns, connected in a unidirectional ring. Each column contains a portion of the issue window, a bank of the physical register file, and an ALU.

The restricted ring topology of RingScalar reduces electrical loading on latency-critical communications between columns, such as instruction wakeup and value bypassing. Since most communication is now kept within a column, the global interconnect overheads and the number of ports required for each component are also reduced. Furthermore, RingScalar exploits the fact that most decoded instructions are waiting on only one operand [EA02, KL03] to use just a single source tag in each issue window entry, and dispatches instructions to columns according to data dependencies to reduce the performance impact of the restricted communication. This RingScalar banking approach reduces the cost of broadcasting tag information across the instruction window, and only requires a single wakeup tag port and tag comparator on each instruction entry. The instruction window banking scheme also meshes smoothly with a banked register file scheme to reduce the number of write ports required on each cell in the physical register file to one.

In section 5.1, RingScalar microarchitectures are described in detail and are compared against the con-

ventional superscalar. The statistics on both operand availabilities and dependencies are presented in Section 5.2 to determine the potential performance impact of implementing a banked instruction issue window. Then, simulated results and circuit complexities of RingScalar are analyzed in Section 5.3 and Section 5.4. To finish the discussion, Section 5.5 summarizes the design and concludes the findings.

5.1 RingScalar Microarchitecture

The RingScalar design builds upon earlier work in banked register files [WB96, CGVT00, BDA01, PPV02, TA03], tag-elimination [EA02, KL03], and dependence-based scheduling [KF96, PJS97]. For clarity, this section describes RingScalar in comparison to a conventional superscalar.

5.1.1 Architecture Overview

RingScalar retains the overall instruction flow of a conventional superscalar and uses the same reorder buffer and memory queue, but drastically reduces the circuitry required in the issue window, register file, and bypass network by restricting global communication within certain structures. These restrictions exploit the fact that most instructions enter the issue window waiting on one or zero operands.

The overall structure of the RingScalar microarchitecture is shown in Figure 5-1. RingScalar divides an N-way issue machine into N columns connected in a unidirectional ring. Each column contains a portion of the issue window, a portion of the physical register file, and an ALU. Physical registers are divided equally among the columns, and any instruction that writes to a given physical register must be dispatched and issued in the column holding the physical register. This restriction means each bank of the regfile needs only a single write port directly connected to the output of the ALU in that column.

A second restriction is that any instruction entering the window while waiting for an operand must be dispatched to the column immediately to the right of the column containing the producer of the value (the leftmost column is considered to be to the right of the rightmost column in the ring). This restriction has two major impacts. First, when an instruction executes, it need only broadcast its tag to the neighboring column which reduces the fanout on the tag wakeup broadcast by a factor of N compared to a conventional window. Second, the bypass network can be reduced to a simple ring connection between ALUs as any other value an instruction needs should be available from the register file. The bypass fanout is reduced by a factor of N, and each ALU output now only has to drive the regfile write port and the two inputs of the following
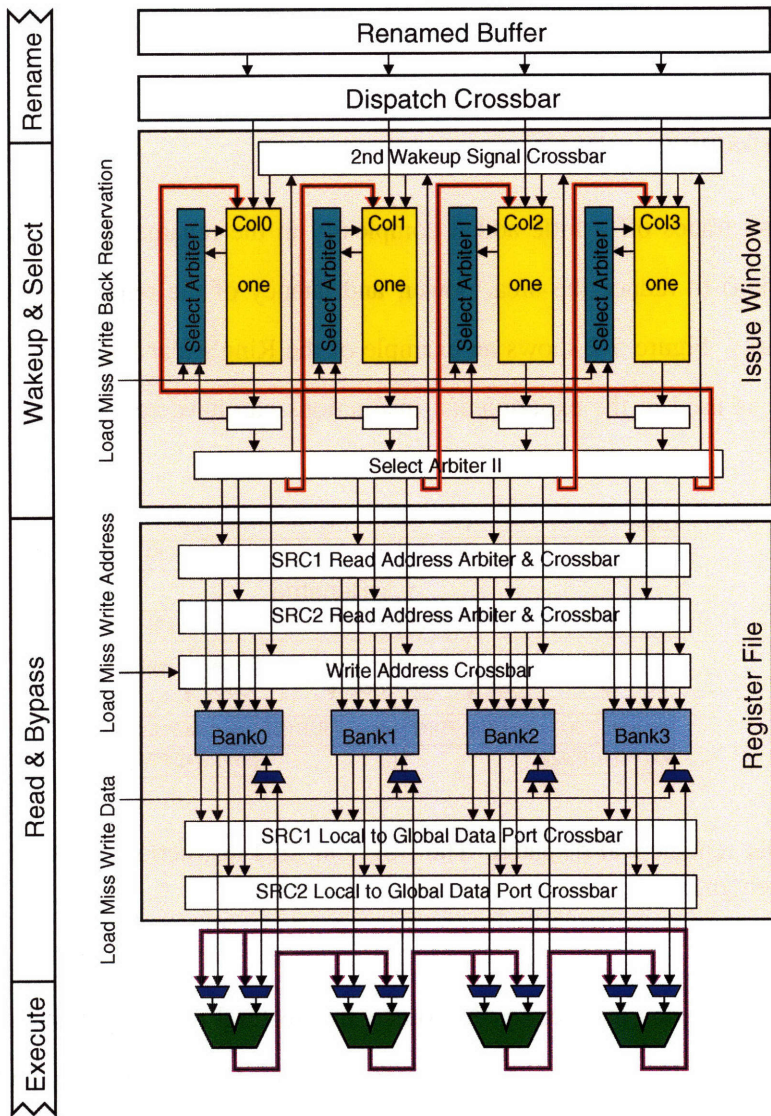
72

Figure 5-1: RingScalar core microarchitecture for a four-issue machine. The reorder buffer and the memory queue are not shown.

ALU.

These restrictions are key to the microarchitectural savings in RingScalar, and as shown in the simulation model, have a relatively small impact on instructions per cycle (IPC) while significantly reducing area, power, and circuit latency. The following subsections describe the major components of the machine in more detail.

5.1.2 Register Renaming

The RingScalar design trades off a little added complexity in the rename and dispatch stage (and some overall IPC degradation) to reduce the area, power, and latency of the remaining stages (issue window, regfile, bypass network). Figure 5-2 shows an example of the RingScalar renaming and dispatch process. Since the Alpha ISA is used in the experiments, instructions can have zero, one, or two source register operands.

Oldest I0	I1	I2	I3
lw r1 (r3)	add r1, r1, #1	sw r1 (r4)	xor r3, r1, #4

⬇ Rename

| lw P1[5], P0[4] | add P2[3], P1[5] | sw P2[3], P3[4] | xor P3[4], P2[3] |

Dispatch Conflict

Col0	Col1	Col2	Col3
sw addr. P3[4]	lw P1[5], P0[4]	add P2[3], P1[5]	sw data P2[3]
sub P0[4], P3[2]			and P3[4], P2[0]
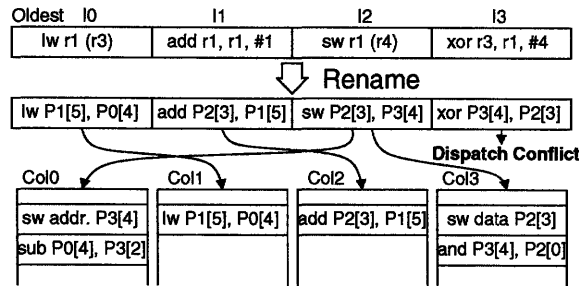
Figure 5-2: RingScalar rename and dispatch. The sub and and instructions were already in the window before the new dispatch group.

As RingScalar decodes each instruction, the source architectural registers are renamed into the appropriate physical registers and the readiness of these operands is checked, just as in a conventional superscalar processor. Instructions that are not waiting on any operand (*zero-waiting*) can be dispatched to any column in the window. Instructions that are waiting on one operand (*one-waiting*) must be dispatched to the column to the right of the producer column. Instructions waiting on two operands (*two-waiting*) are split into two parts that will issue sequentially, and each part must be dispatched to the column to the right of the appropriate producer column. A separate 2nd wakeup port is provided on each column to enable the first part of an instruction to wakeup the second part regardless of the column in which it resides. The second part sits in the window, but will not request issue until after the first part has issued and woken up the second

74

part. Store instructions are handled specially by splitting them into two parts (address and data) that can issue independently and in parallel. Previous work has considered the use of prediction to determine which operand of a two-waiting instruction will arrive last [EA02], but in this work, a simple heuristic that assumes the first (left) source operand will arrive last for a two-waiting instruction is adopted.

To reduce complexity in both the dispatch crossbar and the issue window entries, RingScalar has only a single dispatch port for each column in the window. The two parts of a store instruction or two-waiting instruction occupy two separate dispatch ports if they go to different columns, but can be dispatched together in one cycle to the same column. Note that a physical register tag is now divided into two fields: the window column and the register within the column.

As with a conventional superscalar, the first step of renaming is a rename table lookup to find the current physical register holding each architectural register source operand together with its readiness, while, in parallel, the architectural source registers of later instructions in the group are checked for dependencies on the architectural destination registers of earlier instructions. Then, the column selector assigns the dispatch column for each instruction and the dispatch column arbiter grants the dispatch port to allow each instruction to enter the issue window. The final step of renaming is to allocate a destination physical register for the instruction if required (store and branch instructions do not require destination registers). Any instruction that is granted a dispatch port simply takes the head of the relevant free list because each column has a separate physical register free list.

For dispatch column assignments, the RingScalar renamer has some flexibility in how any zero-waiting instructions (and any dependents) are mapped to columns. A simple greedy scheme where instructions are considered in program order is chosen to minimize the complexity of the rename logic. Zero-waiting instructions select a dispatch column using a random permutation that changes each cycle. One-waiting and two-waiting instructions have no freedom and must be allocated as described above. When the next instruction cannot be dispatched because a required dispatch port is busy, dispatch stalls until the next cycle.

Figure 5-3 shows the renaming and column dispatch circuitry in detail. Each rename table lookup returns the column (*Col*) in which the physical register resides and a single bit (*Rdy?*) indicating if the value is ready or not, in addition to the physical register number. Below the rename table in Figure 5-3, only the circuitry responsible for column allocation is shown, without the physical register number within the column. Column information is represented using a unary format with one bit per window column (i.e., *Col.*
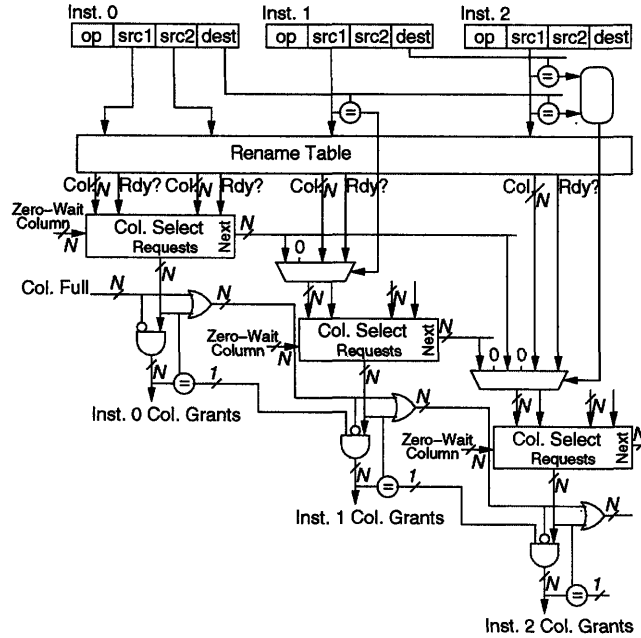
Figure 5-3: RingScalar register renaming and column dispatch circuitry. Only the circuitry for src1 of instruction 1 and 2 is shown.

is an N-bit vector) to simplify circuitry.

For each instruction, the *Col. Select* circuitry calculates two N-bit column vectors: *Requests* and *Next*. The *Requests* vector has one or two bits set indicating which columns the instruction wants to dispatch into, and is calculated in two steps. First, if both of the operands are ready, an internal vector is set to the precomputed *Zero-Wait Column* vector which has a single bit pointing at the randomly assigned column for this instruction. If at least one of the operands is not ready, the internal vector is set to the bitwise-OR of the two input *Col* vectors. The internal vector is then rotated by one bit position to yield the *Requests* vector. The rotation is simple rewiring and so has no additional logic delay.

The *Next* output has a single bit set indicating the column into which this instruction will write its final result. First, the internal vector is assigned either *Zero-Wait Column* if both operands are ready, *Col* for the second source only if the instruction is one-waiting on the second operand, or otherwise *Col* for the first source (i.e., one-waiting on first operand or two-waiting). Second, the internal vector is rotated by one bit position to obtain the *Next* vector.

Any later instruction in the current dispatch group that has a RAW dependency on an earlier instruction in the group must mux in the *Next* vector from the earlier instruction in place of the stale column vector read from the rename table. In the worst case, a series of serially-dependent instructions requires the *Next* values

76

to ripple across the different *Col. Select* blocks in the dispatch group, as shown in Figure 5-3. Fortunately, mux select lines are available early, and the ripple path always carries non-ready operands (*Rdy?*= 0), which reduces worst-case latency to a few gate delays per instruction.

The dispatch column arbiter is implemented using the serial logic gates shown at the bottom of Figure 5-3. The leftmost input to the arbiter chain is a *Col. Full* vector indicating which columns cannot accept a new instruction, either because the issue window is full or because there are no free physical registers left in the column. The arbiter ensures that instructions must dispatch in program order, by preventing later instructions from dispatching if an earlier one did not get all requested columns (this is the purpose of the equality comparators). The ripple through the arbiter is in parallel with the slower ripple through the *Col. Select* blocks, so adds only a single gate delay before yielding the column grant signals.

This additional column latency in RingScalar is compensated by the reduced dispatch latency, as each dispatch port fans out to N times fewer entries than in a conventional superscalar, and each entry has one port rather than N. Also, note that the column allocation circuitry is a small amount of logic (dozens of gates on each of the N bit slices) and represents a very small power and area overhead compared to the savings in issue window and register file size.

5.1.3 Issue Window

The RingScalar issue window has several complexity reductions compared to a conventional superscalar. The primary savings come from the reduced port count in each column. A conventional superscalar window has N dispatch ports, $2N$ wakeup ports, and N issue ports on each entry. RingScalar has only a single dispatch port, two narrower wakeup ports, and one issue port.

Each column needs only two wakeup ports. The first wakeup port is used by the preceding column to wake up dependent instructions, while the second port wakes up the second part of a two-waiting instruction. Both of these ports are narrower than in a conventional superscalar as shown in Figure 5-4. The physical register tag requires $\lg(N)$ fewer bits because the consumer must be waiting for a value located in the preceding column. The second-part tag can be considerably narrower as it only needs to distinguish between multiple second parts mapped to the same column in the issue window.

The RingScalar instruction window design significantly reduces the critical wakeup-select scheduling loop. Wakeup has reduced latency because an instruction only has to broadcast its tag to one column, each
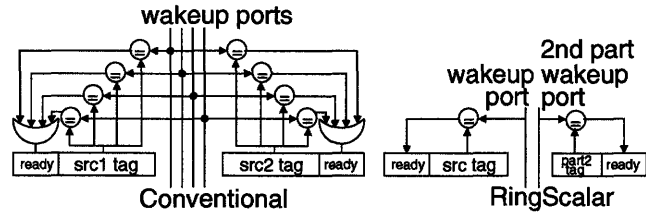
77

Figure 5-4: Wakeup circuitry.

entry has only one comparator, and each tag is narrower. A conventional design requires the tag be driven across the entire window, with each entry having two comparators, leading to a $2N$ times greater fanout in total. The select arbiter also has considerably reduced latency, as each column has a separate arbiter, and each column can only issue at most one instruction. The conventional design has an arbiter with N times more inputs and N times more outputs.

Each entry of RingScalar has only a single issue port, which reduces electrical loading to read out instruction information after select. A conventional design has each entry connected to N issue ports, each with N times greater fanout. We implement a two-level select arbitration scheme for RingScalar. First, each column has an independent local arbiter that selects one instruction to issue from the ready instructions on that column. This arbiter is $o(n^2)$ less complex than the global arbiter, as it has $1/N$ the number of request inputs and $1/N$ the number of grant outputs. An N-way arbitration is then performed among the selected instructions for execution resource constraints.

The combination of a single dispatch port and a single issue port makes it particularly straightforward to implement a compacting instruction queue [BAB$^+$02], where each column in the window holds a stack of instructions ordered by age with the oldest at the bottom (Figure 5-5). Instead of having input ports connected to $2N$ fan-in mux and output ports with $2N$ fanout, each entry now reduces the fan-in and fan-out by a factor of N. Similarly, the electric loads to each dispatch port and issue port is decreased by a factor of N because the entries are spread evenly across N columns. Another benefit of RingScalar issue window is its simplification in the compaction multiplexing circuitry, results in reduced area.

The instructions are latched after issue, then undergo a second stage of select arbitration (*Select Arbiter II* in Figure 5-1) that is used to resolve structural hazards across columns. For example, the baseline machine only allows a single load to issue per cycle. RingScalar also allows only a single first-part sub-instruction to wake-up a second-part sub-instruction across the *2nd Wakeup Signal Crossbar* each cycle to minimize
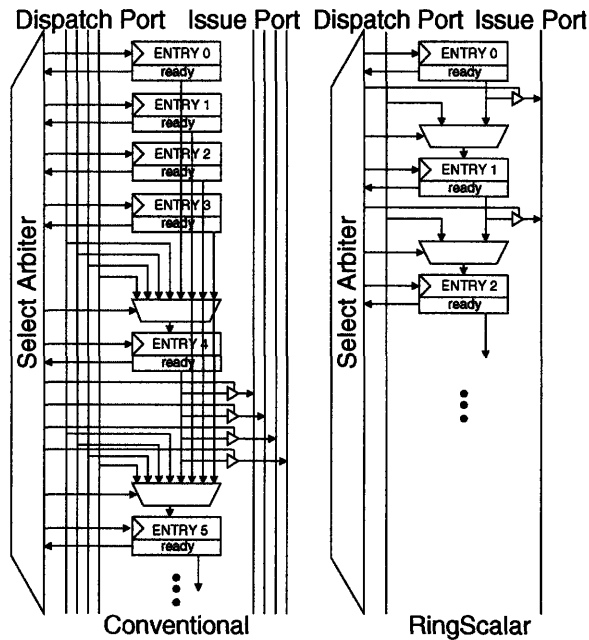
Figure 5-5: Latch-based compacting instruction queues.

circuitry. Instructions failing the second stage of arbitration remain in the issue latch and block further issue in the column until the structural hazard is resolved.

In practice, instruction entries are not compacted out right after issue, but only after it is known they will complete successfully. In particular, dependent operations are scheduled assuming loads will hit in the cache and must remain in the window until the cache access is validated in case a miss requires the dependent instruction be replayed. As described in the following section, RingScalar uses the same technique if a banked register file with read conflicts is used.

5.1.4 Banked Register File

One of the greatest savings in RingScalar comes from the reduction in the number of write ports required on the register file. Each column has a separate physical register bank, which needs only a single write port. RingScalar allows any column to read data from any register bank in the machine but only allows to write data to its register bank. In the simplest configuration, we provide a full complement of read ports $(2N)$ on every bank. To further reduce regfile power and area, the lesser-ported banked regfile presented in Chapter 4 can be used. For example, the four-issue machine shown in Figure 5-1 has four read ports and one write port per bank whereas a conventional machine would have eight read ports and four write ports.

79

The speculative control scheme is adopted to handle the reduced-read-port design without adding complexity to the issue logic. Instructions continue issuing assuming there will be no conflicts. When read-port conflicts are detected, the instruction window must be repaired and execution must be replayed. The number of read bank conflicts is reduced by not requesting read port accesses when a value was produced in the preceding cycle and hence will be available on the bypass ring (*conservative bypass-skip*), and by implementing the *read-sharing* optimization, which allows a single bank port to send the same register to multiple requesters over the global ports.

As mentioned in the previous chapter, the speculative control design adds an additional arbitration stage to the pipeline as shown in Figure 4-5. A local to global read port crossbar is also required to allow any functional unit to read any register from any local bank's read port. Unlike the design in Chapter 4, there is no need for a global write port network and an arbiter with bank conflict detection, as the RingScalar design associates one column with each write port and issues at most one instruction per column. This is a considerable saving, as it was also previously found that more than one write port per bank was required to reduce write port conflicts to an acceptable level. However, variable latency instructions, such as cache misses, still require access to the register file write ports to return their results. To avoid write conflicts, a returning cache miss inserts a high priority request into the select arbiter for the target column, preventing another instruction from issuing while the cache miss uses the write port.

5.1.5 Bypass Network

RingScalar also provides a large reduction in bypass network complexity. An ALU can only bypass to its neighbor around the ring, not even to its own inputs. This bypass path is sufficient because the dependence-based rename and dispatch ensures dependent instructions are located immediately following the producer in the ring. If an operand was ready when the instruction was dispatched, it would have been obtained from the register file in any case. If a dependent instruction does not issue right after the producer, it must wait until the value is available in the regfile before issuing. Comparing to the fully-bypassed superscalar, RingScalar reduces each ALU fanout by $(N - 1)$.
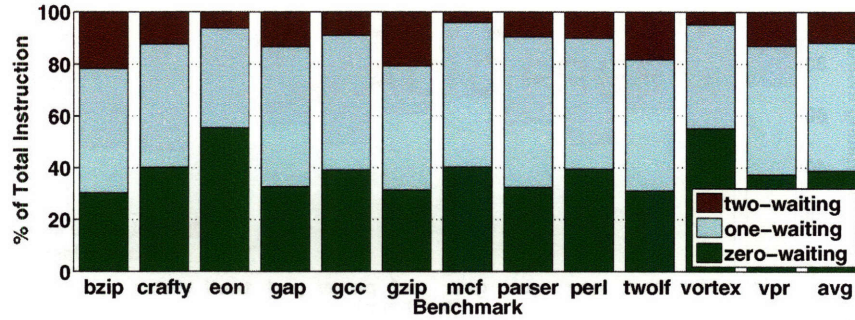
Figure 5-6: Percentage distribution of zero-waiting, one-waiting, and two-waiting instructions.

5.2   Operand Availability

Previous work indicates that issue window source tags are underutilized and many instructions enter the issue queue with only zero or one outstanding register operands [EA02, KL03]. For this reason, RingScalar halves the number of wakeup tags in the issue window. In RingScalar, instructions with two unmet source operands at the time of dispatch are split into two parts and occupy two entries, which affects the overall capacity of the instruction window. To confirm the observation that more than 80% of instructions wait on only one or zero operand, Figure 5-6 shows the distribution of zero-waiting, one-waiting, and two-waiting instructions on our four-way baseline superscalar processor running the SPEC CINT2000 benchmark suite. The percentage of two-waiting instructions ranges from 3.9% (mcf) to 21.9% (bzip) with an average of 11.8%.

As the second part of a two-waiting instruction will only be issued if woken up by the first part, the arrival timing of the two unmet operands impacts the performance of RingScalar. When the source operand of the second part arrives last, the waking of the second part is likely to finish before the instruction is ready to be issued. However, if the second part arrives first, the waking of the second part delays issue of the ready instruction for additional cycles. RingScalar uses a simple scheme where the left source operand is always predicted to arrive last. Figure 5-7 shows that the left source operand arrives last more than half of time in eight out of twelve programs (ratio ranges from 46.7% to 71.3% with an average of 56.4%). These numbers do not include store instructions, where the address calculation and data movement issue independently and in parallel.
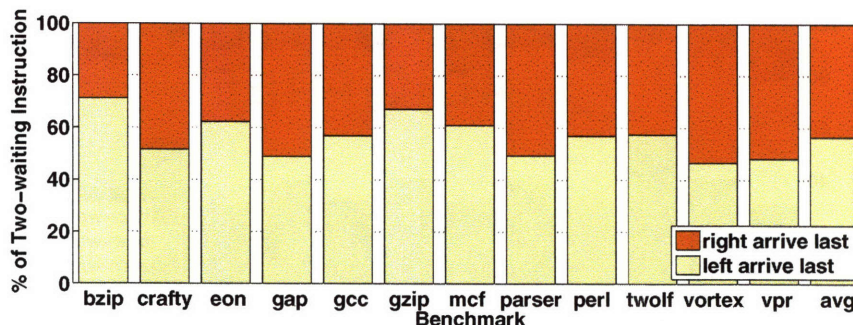
Figure 5-7: Percentage distribution of last-arrival operand for two-waiting instructions.

5.3 Evaluation

RingScalar reduces the area, latency, and power of all major structures in the instruction flow by dividing an N-way superscalar into N columns connected in a unidirectional ring. This distributed hardware structure reduces the utilization of hardware resources and requires a larger number of register file and issue window entries, which is analyzed in Section 5.3.1. Next, the performance of RingScalar in comparison to the conventional design is evaluated in Section 5.3.2. Section 5.3.3 analyzes the effects RingScalar has on regfile read port reduction techniques. Section 5.3.4 presents the IPC counts of an alternative RingScalar design where a separate issue widow is added to handle two-waiting instructions.

In this section, each configuration is labeled with the following nomenclature: *(arch)(#iq):(size)R(#read)W(#write)*, where *(arch)* is either the monolithic baseline *(BL)* or the RingScalar *(RS)* architecture, *(#iq)* is the total number of instruction window entries, *(size)* defines the regfile size, *(#read)* and *(#write)* are the number of read ports and the number of write ports in each regfile storage cell. An ideal 4-issue superscalar configuration *BL32:80R8W4* is selected to provide baseline numbers. It contains a conventional monolithic issue window with 32 issue queue entries, a fully multiported register file with 80 registers, and a full bypass network. For RingScalar, entries are evenly distributed among columns. For example, *RS48:128R8W1* is a RingScalar design where each of the four columns has twelve issue window entries, and a bank of 32 registers with eight read ports and one write port. Any RingScalar with a speculatively-controlled banked register file, has an additional read port arbitration stage added as shown in Figure 4-5, and the branch misprediction penalty is increased by one cycle.
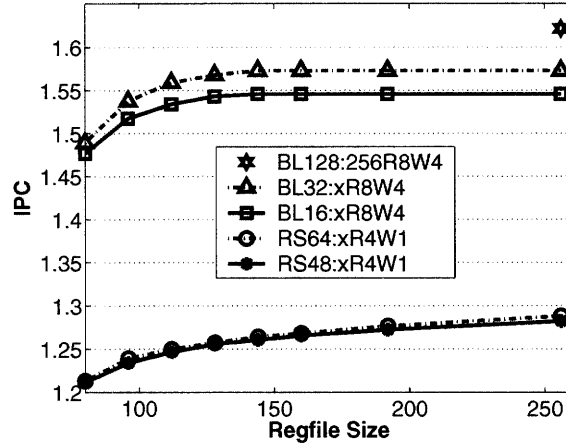
82

Figure 5-8: Average IPC comparison for different regfile sizes.

5.3.1 Resource Sizing

The register file and issue window of RingScalar is spread evenly across the columns. Their utilization is less than a monolithic structure and so the optimal sizing needs to be re-evaluated. The first set of experiments determines the effect of increasing the regfile size while keeping the issue window fixed. The second set of experiments finds out how performance varies when increasing the size of the RingScalar issue window.

Figure 5-8 shows diminishing performance improvements as the regfile size is increased for both the baseline superscalar *BL32:xR8W4* and the RingScalar *RS48:xR4W1* processor. For the baseline design, IPC saturates at 144 registers; performance remains the same if additional registers are added beyond this point. RingScalar, however, keeps improving as more registers are added. This is because instructions can only be allocated to a particular column in RingScalar, and an imbalance of registers across the columns lowers the total regfile utilization. Nevertheless, the diminishing returns does not justify implementing a regfile that is larger than 128 for issue queue sizes of 48 and 64. The performance of the BL128:256R8W4 configuration is also plotted to show the limit on IPC for these codes with this simulation framework.

Performance increases as the issue window size grows as shown in Figure 5-9. For designs with 256 registers, IPC improvement tapers off beyond a window size of 64; for designs with 128 registers, it tapers off beyond a 48-entry window. This also demonstrates the significance of a balanced design, increasing the resource in just a single area will not always lead to higher performance. Therefore, RS64:256 and RS48:128 are chosen as the two basic parameter settings for the RingScalar evaluation.
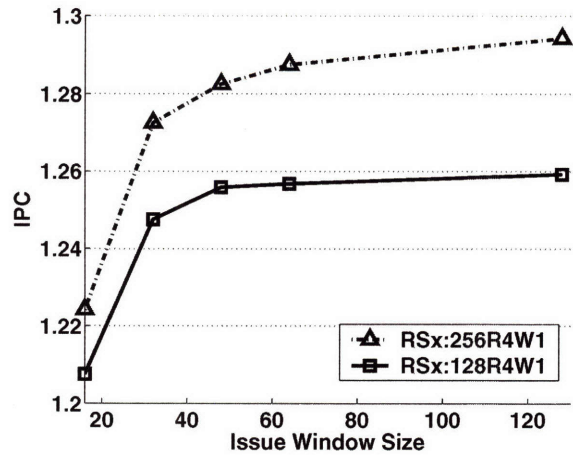
Figure 5-9: RingScalar average IPC sensitivity to instruction window size.
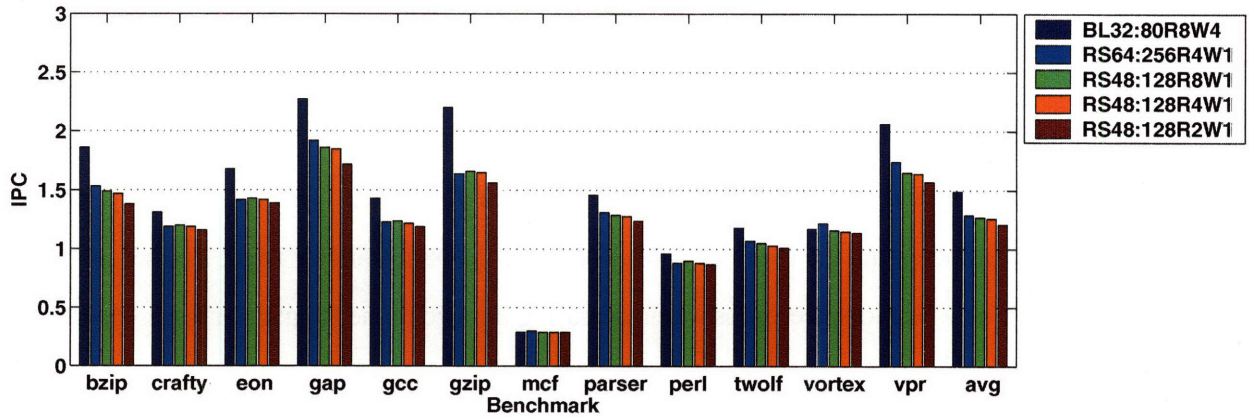
Figure 5-10: IPC for 1 thread workload with a gshare branch predictor.

5.3.2 IPC Comparison

To evaluate RingScalar, its performance is compared against the baseline configuration. Simulations were run with a gshare branch predictor and with a perfect branch predictor to ascertain the effect of the extra pipeline stage and branch predictor inaccuracies. Figure 5-10 shows the absolute IPC obtained with a gshare branch predictor, while Figure 5-11 shows the results using perfect branch prediction. IPCs increase 12%, on average, for designs where the processors only fetch and execute code on the correct path. The relative performance differences between the baseline and RingScalar remains consistent across different branch predictor precisions.

Overall, the results for the RingScalar design are quite competitive with the idealized superscalars given

Figure 5-11: IPC for 1 thread workload with a perfect branch predictor.

their simplified window design, and their much more realistic implementation parameters. In comparison to the baseline (*BL32:80R8W4*), the performance of the small RingScalar design without regfile read-port conflicts (*RS48:128R8W1*) has an average of 12% IPC reduction and a maximum degradation of 24%. The performance impact is mainly due to delayed issuing of critical instructions, which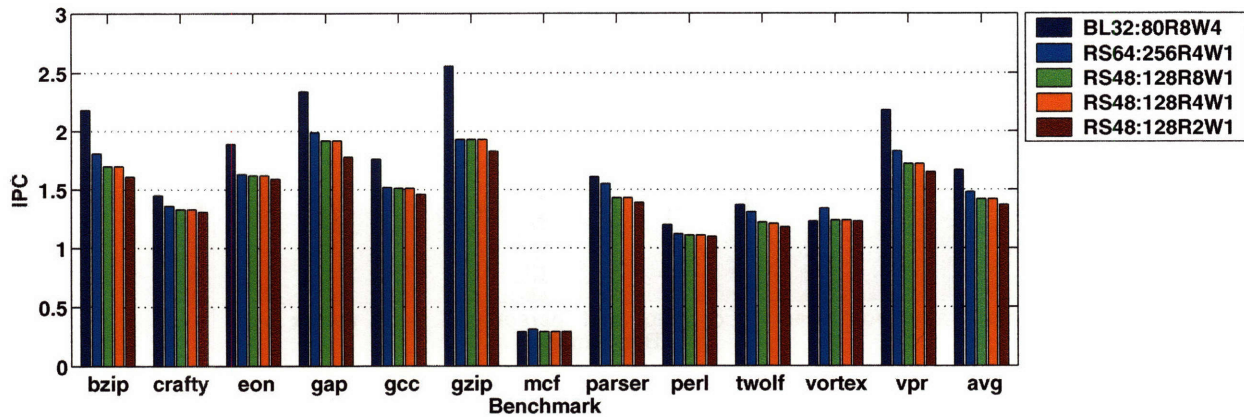 can pile up in the same issue column. Extra regfile savings can be achieved in RingScalar with a lesser-ported banked structure. Figure 5-10 show that IPC drops only another 1% for *RS48:128R4W1* design but 4% for *RS48:128R2W1* design. Further comparing the *RS48:128R4W1* design to the large RingScalar (*RS64:256R4W1*) design, only a 2% IPC difference is observed. The above data supports that *RS48:128R4W1* is a good design point for a four-issue machine.

5.3.3 Regfile Read Port Optimization Effectiveness

The two types of optimization, conservative bypass-skip and read-sharing, decrease the number of regfile bank conflicts in the lesser ported banked structure by reducing the number of read port contentions. If an instruction is selected in the same cycle where a tag match caused the instruction to wake up, the conservative bypass-skip scheme avoids competing for the regfile read port. In the baseline machine, on average, 36% of the total read port conflicts are avoided when conservative bypass-skipped is implemented, as shown in Figure 5-12. This percentage drops to only 23% for the RingScalar design because RingScalar steers instructions that wait on the same operand into the same column and only allows a single instruction to be issued per column each cycle. Instructions that normally read values from the bypass network now have to
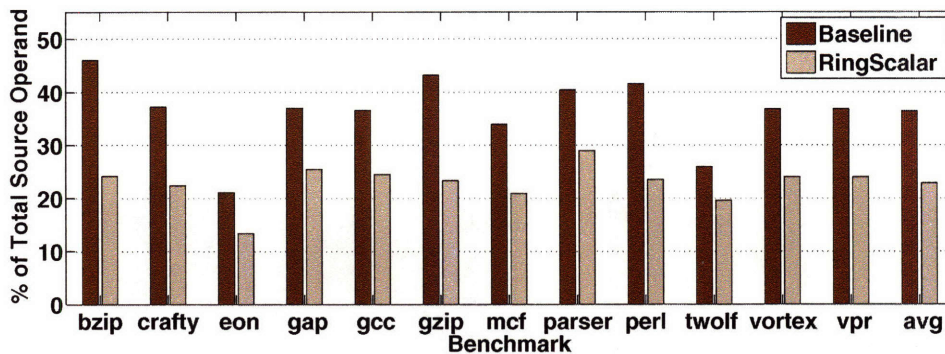
Figure 5-12: Percentage of operands that do not compete for regfile read port due to conservative bypass-skipped optimization.
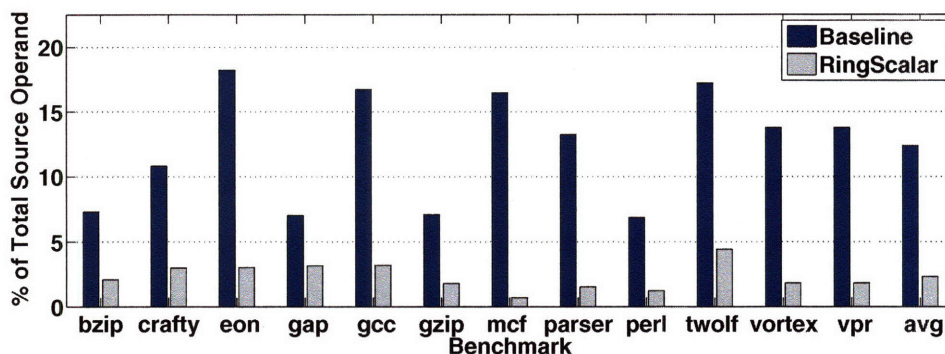
Figure 5-13: Percentage of operands that do not compete for regfile read port due to read-sharing optimization.

access the data from the regfile because of their delayed issue. The use of the conservative bypass-skipped optimization is not as effective in RingScalar as in the baseline design but it still avoids a significant number of read port conflicts.

Conservative bypass-skip screens out instructions that depend on the same register and are issued in the same cycle when instructions become ready. The read-sharing optimization further reduces the number of bank conflicts by removing read port contention for instructions that depend on the same register but are not issued in the same cycle as their tag match. Figure 5-13 shows the percentage of reads that avoid accessing the regfile if we allow a local port to drive multiple global ports. The average result across the benchmark suite is 12% for the baseline superscalar and 2% for RingScalar. Again, this is a result of the way RingScalar dispatches instructions into columns and its restriction on the number of instructions that can be issued per column on each cycle. Instructions that share the same dependent registers would be packed into the same

column and would never be issued together if the data is not ready at the time of dispatch. In RingScalar, read-sharing can only benefit a small number of instructions that depend on the same ready register and that are issued in the same cycle.

5.3.4 Two-waiting Queues

The performance loss of using only a single-tag instruction window is evaluated by comparing the results to a variation of a RingScalar design where each issue queue column is further divided into three banks. Figure 5-14 shows that instruction banks are of three types, depending on whether instructions are waiting on zero, one, or two source register operands. Unlike the original RingScalar design, instructions that wait on both operands that are placed in the banks with two source tags. The two-waiting queues reduces the complexity in register renaming logic and eliminates the prediction on operand availability as two-waiting instructions are no longer split into two parts. Simulated results show that a 3% IPC improvement is observed across the benchmarks after a 16-entry two-waiting queue is added to RingScalar (*RS48:128R4W1*). This small enhancement does not justify the additional area and power consumption of two-waiting queues.

5.4 Complexity Analysis

To determine the complexity effectness of RingScalar designs, the area, latency, and power reductions of key components are analyzed in this section. The approach is to first compare required regfile die area by counting the number of occupied wire tracks. Then, the factors that determine latency and power of issue windows are evaluated.

The area of a register file can be estimated by the number of bitlines and the number of word-lines [RDK+00, TA03]. For regfiles with single-ended reads and differential writes, Equation 5.1 is used to approximate the area for each bit-slice. The width (w) and the height (h) of a storage cell, including power and ground, is given in unit of wire tracks. R is the number of read ports, W is the number of write ports, and E is the number of entries in the regfile. The equation expresses that each regfile port calls for one wordline per entry, plus a single bitline for read ports and a pair of bitlines for write ports.

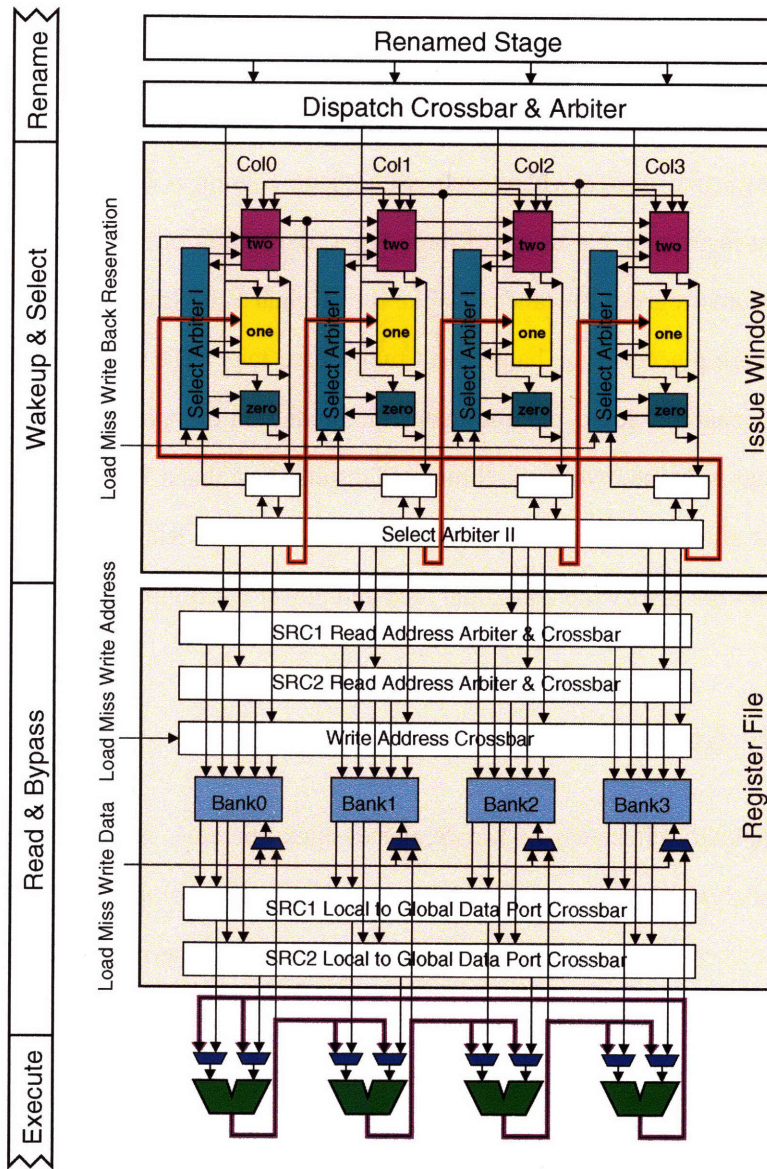$$Area_{Regfile} = (w + R + W) \times (h + R + 2W) \times E \qquad (5.1)$$

Figure 5-14: RingScalar architecture for designs with three issue banks per column.

An issue window consist of dispatch ports, issue ports, wakeup port, comparators, tag broadcast network, and select arbiters. The speed of the wakeup-select loop is a function of wire propagation delay and the fan-in/fan-out delay. Its power consumption is proportional to switched capacitance, such as wire capacitance and transistor parasitic capacitance. RingScalar reduces these parameters by adopting lesser-ported banked structures. Power saving can also be achieved by minimizing the number of active components, such as comparators. Using Equation 5.2, the number of bit comparators in an issue window can be determined. T is the number of tags per entry, B is the number of tag bits per entry (depends on the regfile size), Y is the number of wakeup port per tag, and E is the number of entries. For example, there are $2 \times 7 \times 4 \times 32 = 1792$ bit-comparators in the baseline (*BL32:80R8W4*) design.

$$Number_{bit-comparators} = T \times B \times Y \times E \qquad (5.2)$$

Table 5.1 provides a complexity and performance comparison across a few RingScalar designs and the baseline. In general, RingScalar designs are smaller and more power efficient than the conventional superscalar. Even the large RingScalar design (*RS64:256R4W1*) has a smaller regfile, a much simpler issue window (faster wakeup-select), and a smaller bypass network than the baseline, despite the increased number of regfile entries.

The *RS48:128R4W1* design point appears to be a sweet spot in performance-complexity. Regfile area is under half that of the baseline while the issue window is much smaller, and IPC is just 13.3% away from the baseline. Adding more read ports only increases IPC by 1% (*RS48:128R8W1*). The additional regfile area saving of moving from *RS48:128R4W1* to *RS48:128R2W1* is only 9% but causes a 3% drop in IPC. Furthermore, a 48-entry RingScalar issue window requires only one fourth the number of dispatch, issue, and wakeup ports, and 21% of the tag comparators of a conventional 32-entry design. Table 5.1

Configuration	Regfile Area	Issue Window		Wakeup Fan-out	Select Arbiter	Bypass Networks		IPC
		# Dispatch/Issue/ Wakeup Ports	# Compa- rators			ALU Fan-out	MUX Fan-in	
BL32:80R8W4	100.0%	4/4/8	100.0%	64	4 from 32	9	7	100.0%
RS64:256R4W1	80.8%	1/1/2	28.6%	16	1 from 16	3	4	89.7%
RS48:128R8W1	87.6%	1/1/2	21.4%	12	1 from 12	3	4	87.6%
RS48:128R4W1	40.4%	1/1/2	21.4%	12	1 from 12	3	4	86.7%
RS48:128R2W1	31.4%	1/1/2	21.4%	12	1 from 12	3	4	84.2%

Table 5.1: Total complexity comparisons. Percentage results are normalized to the baseline (BL32:80R8W4).

also shows that RingScalar reduces wakeup delay because of its reduction in the wakeup broadcast fan-out. Comparing to the baseline configuration, *RS48:128R4W1* has faster select timing. *BL32:80R8W4* has one arbiter that selects four instructions out of a pool of 32 instructions while RingScalar has four arbiters, each independently selecting only one out of 12 instructions. The reduced bypass network decreases the ALU fan-out by a factor of three while the bypass mux fan-in is cut from seven to four for the RingScalar bypass networks.

In this evaluation, the baseline architecture was idealized in several respects. More realistic superscalar models should have a reduced IPC advantage over RingScalar. For example, most designs approximate the oldest-first select arbitration to reduce circuit complexity; real designs have less than fully orthogonal functional unit issue; and they will experience load-hit mispredictions and memory dependence misspeculations. These additional stalls will tend to reduce the IPC advantage of existing superscalar designs. The complexity reduction of RingScalar could enable overall performance improvement by reducing circuit latency, and hence clock frequency, and also power, while reduce considerable area.

5.5 Summary

The RingScalar design builds upon earlier work in banked register file, tag-elimination, and dependence-based scheduling. It provides complexity reduction throughout the major components of an out-of-order processor, in exchange for a small increase in complexity in the rename and dispatch stage. Compared with idealized superscalar architectures, there is only a small (10.3-13.3%) drop in IPC but with a large reduction in area, power, and latency of the issue window, register file, and bypass network. RingScalar should be even more competitive against realistic conventional superscalar processors, and should provide a suitable design point for CMP cores that need both high single thread performance and lower power and area.

Chapter 6

Conclusion and Future Work

I have shown that banking works well compared to monolithic structures. A banked register file design can provide the bandwidth needed by a superscalar processor but with reduced area, delay and energy. For a quad-issue processor, I show that banking reduces regfile size by over a factor of three, access time by 25% and access energy by 40%, while reducing IPC by 6%. I also presented the RingScalar architecture based on a ring topology of banked structures that reduce the complexity of dynamically scheduled superscalar processors. The design exploits the fact that most decoded instructions are waiting on just one operand to use only a single tag per issue window entry, and to restrict instruction wakeup and value bypass. It limits the resources required in all stages of execution, from dispatch, wakeup, select, issue, regfile read, bypass, and write bank. For four-issue processors, IPC drops 13% but with large complexity reduction that can be used to increase clock rate, leading to a more complexity-effective design.

6.1 Summary of Contributions

The main contributions presented in this thesis are classified into two categories:

Banked Regfile

- *The design of full-custom banked register files.* I show that the multiplexing circuits dominate the area of few-ported multibanked design. Moving from a single read port to split dual read ports per bank only impacts the area minimally, yet significantly reduces control logic complexity.

- *The invention of a conservative bypass-skip technique that uses wakeup tag search to determine bypassability of source registers.* The bypass bit is only set if the bypass will occur from the immediately preceding cycle. This scheme does not save register port bandwidth for operands that will be bypassed from later bypass stages but can be easily implemented with only one bit per register read.

- *The invention of a speculative control scheme that issues instructions without considering bypassability or register file bank conflicts.* It relies on rapid port arbitration and a non-stalling pipeline repair to reduce the pipeline complexities in handling limited regfile ports.

RingScalar

- *The design of a banked issue window that uses only single-tag entries and places instructions into columns according to the dependency.* I also show that the small performance lost in single-tag banked issue window structure can be well justified by its large complexity reductions.

- *The design and evaluation of RingScalar—a complexity-effective banked superscalar microarchitecture.* The design divides an N-way superscalar into N columns connected in a unidirectional ring, where each column contains a portion of the instruction window, a bank of the register file and an ALU. I show that RingScalar simplifies all the major components in the instruction flow to increase area and power efficiency without excessive pipeline control complexity.

6.2 Future Work

There are several avenues of research which lead on from this work:

Banked Reorder Buffer Structure

Although this thesis discusses only banked register files and banked issue windows, the banking techniques can be applied to other centralized SRAM/CAM-like structures, such as the ROB. For example, the number of write ports can be minimized if the ROB is also divided into columns as in the rest of the RingScalar architecture.

Banked Load-Store Queue

There are many recent papers [CE98, POV03, SMR05] concerning the complexities in load-store queues. Since a load-store queue is another centralized SRAM/CAM-like structure, banking techniques [GAR$^+$05, BZ06] can also be applied to reduce its die area, access time, and power. The difficulty in banking this structure is maintaining the required ordering of elements across banks.

Optimizing Register Renamer

Most machines use SRAM structure to store the mapping between the logical and the physical registers. As the register file size increases, the register renaming hardware overhead also increases. With a banked register file and RingScalar's restricted ring topology, the renaming table can be potentially optimized to reduce circuit complexity.

Adaptive Embedded Cores

Embedded applications usually require only real time performance and emphasis on power consumption. Savings can be achieved if the machine is reconfigurable in-flight according to its workload. Since individual bank of a banked structure can be deactivated with proper techniques, the RingScalar design is a good candidate for power-sensitive embedded cores.

Bibliography

[AF03]   A. Aggarwal and M. Franklin. Energy efficient asymmetrically ported register files. In *ICCD 2003*, pages 2–7, San Jose, CA, October 2003.

[AG05]   J. Abella and A. Gonzalez. Inherently workload-balanced clustered microarchitecture. In *IPDPS-19*, Long Beach, CA, April 2005.

[AKSB01] A. Alvandpour, R. Krishnamurthy, K. Soumyanath, and S. Borkar. A low-leakage dynamic multi-ported register file in 0.13 μm CMOS. In *ISLPED'01*, pages 68–71, Huntington Beach, CA, August 2001.

[AMD99]   AMD. AMD Athlon processor technical brief. AMD Techical Report, December 1999.

[BAB$^+$02] A. Buyuktosunoglu, D. Albonesi, P. Bose, P. Cook, and S. Schuster. Tradeoffs in power-efficient issue queue design. In *ISLPED'02*, pages 184–189, Monterey, CA, August 2002.

[BDA01]   R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *MICRO-34*, pages 237–248, Austin, TX, December 2001.

[BPN03]   K. M. Buyuksahin, P. Patra, and F. N. Najm. Estima: An architectural-level power estimator for multi-ported pipelined register files. In *ISLPED'03*, pages 294–297, Seoul, Korea, August 2003.

[BS03]   S. Balakrishnan and G. Sohi. Exploiting value locality in physical register files. In *MICRO-36*, pages 265–276, San Diego, CA, December 2003.

[BTME02] E. Borch, E. Tune, S. Manne, and J. S. Emer. Loose loops sink chips. In *HPCA'02*, pages 299–310, Boston, MA, February 2002.

[BZ06]   I. Baugh and C. Zilles. Decomposing the load-store queue by function for power reduction and scalability. *IBM Journal*, 50(2):287–297, March 2006.

[CBF00]   A. Chandrakasan, W. J. Bowhill, and F. Fox. *"Design of High Performance Microprocessor Circuits"*. IEEE Press, 2000.

[CE98]   G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *ISCA-25*, pages 142–153, Barcelona, Spain, June 1998.

[CGVT00] J.-L. Cruz, A. Gonzalez, M. Valero, and N. P. Topham. Multiple-banked register file architectures. In *ISCA-27*, pages 316–325, Vancouver, Canada, June 2000.

[Cor89]   Unisys Corporation. Scientific processor vector file organization. U.S. Patent 4,875,161, October 1989.

[DEC90]   DEC. Vector register system for executing plural read/write commands concurrently and independently routing data to plural read/write ports. U.S. Patent 4,980,817, December 1990.

[EA02]   D. Ernst and T. Austin. Efficient dynamic scheduling through tag elimination. In *ISCA-29*, pages 37–46, Anchorage, AK, May 2002.

[EHA03]   D. Ernst, A Hamel, and T. Austin. Cyclone: A broadcast-free dynamic instruction scheduler with selective replay. In *ISCA-30*, pages 253–262, San Diego, CA, June 2003.

[FCJV97] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. G. Vranesic. The Multicluster architecture: Reducing cycle time through partitioning. In *MICRO-30*, pages 149–159, Research Triangle Park, NC, December 1997.

[FF98]   J. A. Farrell and T. C. Fischer. Issue logic for a 600-mhz out-of-order execution microprocessor. *Journal of Solid-State Circuits*, 33(5):707–712, May 1998.

[FGK+02] E. S. Fetzer, M. Gibson, A. Klein, N. Calick, C. Zhu, E. Busta, and B. Mohammad. A fully-bypassed 6-issue integer datapath and register file on an Itanium microprocessor. *IEEE Journal Solid-State Circuits*, 37(11):1433 – 1440, November 2002.

[Fis83]   J. A. Fisher. Very long instruction word architectures and the ELI-512. In *ISCA-10*, Stockholm, Sweden, June 1983.

[GAR+05] A. Gandhi, H. Akkary, R. Rajwar, S. T. Srinivasan, and K. Lai. Scalable load and store processing in latency tolerant processors. In *ISCA-32*, pages 446–457, Madison, WI, June 2005.

[HBHA02] S. Heo, K. C. Barr, M. Hampton, and K Asanovic. Dynamic fine-grain leakage reduction using leakage-biased bitlines. In *ISCA-29*, pages 137–147, Anchorage, AK, May 2002.

[Hen00]   J. L. Henning. SPEC CPU2000: measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, July 2000.

[HP02]   J. L. Hennessy and D. A. Patterson. *"Computer Architecture"*. Morgan Kaufmann, May 2002.

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

[JRB+98] S. Jourdan, R. Ronene, M. Bekerman, B. Shomar, and A. Yoaz. A novel renaming scheme to exploit value temporal locality trhough physical register reuse and unification. In *MICRO-31*, pages 216–225, Dallas, TX, November 1998.

[Kes99] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[KF96] G. A. Kemp and M. Franklin. PEWs: A decentralized dynamic scheduler. In *ICPP'96*, pages 239–246, Bloomingdale, IL, August 1996.

[KL03] I. Kim and M. Lipasti. Half-price architecture. In *ISCA-30*, pages 28–38, San Diego, CA, June 2003.

[KM03] Nam Sung Kim and Trevor Mudge. Reducing register ports using delayed write-back queues and operand pre-fetch. In *ICS-17*, pages 172–182, San Francisco, CA, June 2003.

[KMAC03] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway. The AMD Opteron processor for multiprocessor servers. *Micro*, 23(2):66–76, March 2003.

[Kro81] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA-8*, pages 81–87, May 1981.

[Nag75] L. Nagel. SPICE2. Technical Report ERL-M520, ERL Technical Memo, University of California, Berkeley, 1975.

[PJS97] S. Palacharla, N. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *ISCA-24*, pages 206–218, Denver, CO, June 1997.

[PKE+03] D. Ponomarev, G. Kucuk, O. Ergin, K Ghose, and P. Kogge. The Alpha 21264 microprocessor. *IEEE Transactions on Very Large Scale Integration Systems*, 11(5):789–800, October 2003.

[POV03] I. Park, C Ooi, and T. N. Vijaykumar. Reducing design complexity of the load/store queue. In *MICRO-36*, pages 411–422, San Diego, CA, December 2003.

[PPV02] I. Park, M. D. Powell, and T. N. Vijaykumar. Reducing register ports for higher speed and lower energy. In *MICRO-35*, pages 171–182, Istanbul, Turkey, November 2002.

[Pre02] R. P. Preston et al. Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading. In *ISSCC Digest and Visuals Supplement*, pages 266–500, San Francisco, CA, February 2002.

[RBR02] S. Raasch, N. Binkert, and S. Reinhardt. A scalable instruction queue design using dependence chain. In *ISCA-29*, pages 318–329, Anchorage, AK, May 2002.

[RDK+00] S. Rixner, W. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens. Register organization for media processing. In *HPCA*, pages 375–386, Toulouse, France, 2000.

[RF98]   N. Ranganathan and M. Franklin. An empirical study of decentralized ILP execution models. In *ASPLOS-8*, San Jose, CA, October 1998.

[RJSS97] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith. Trace processors. In *MICRO-30*, pages 138–148, Research Triangle Park, NC, December 1997.

[Rus78]   R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, January 1978.

[SBV95]   G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA-22*, pages 414–425, Santa Margherita Ligure, Italy, June 1995.

[SC00]   S. Sair and M. Charney. Memory behavior of the SPEC2000 benchmark suite. Technical report, IBM Research Report, Yorktown Heights, New York, October 2000.

[SJ01]   P. Shivakumar and N. P. Jouppi. CACTI 3.0: an integrated cache timing, power, and area model. Technical report, Western Research Laboratories, 2001.

[SMR05]   T. Sha, M. M. Martin, and A. Roth. Scalable store-load forwarding via store queue index prediction. In *MICRO-38*, pages 159–170, Barcelona, Spain, November 2005.

[TA00]   J. Tseng and K. Asanović. Energy-efficient register access. In *Proc. of the 13th Symposium on Integrated Circuits and Systems Design*, pages 377–382, Manaus, Brazil, September 2000.

[TA03]   J. Tseng and K. Asanović. Banked multiported register files for high-frequency superscalar microprocessors. In *ISCA-30*, pages 62–71, San Diego, CA, June 2003.

[TA05]   J. Tseng and K. Asanović. A speculative control scheme for an energy-efficient banked register file. *IEEE Transactions on Computers*, 54(6):741–751, June 2005.

[TDF+01] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. IBM Technical White Paper, October 2001.

[TEL95]   D. M. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA-22*, pages 392–403, Santa Margherita Ligure, Italy, June 1995.

[TJS95]   M. Tremblay, B. Joy, and K. Shin. A three dimensional register file for superscalar processors. In *HICSS*, pages 191–201, Maui, HI, January 1995.

[Tom67]   R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal*, 11(1):25–33, January 1967.

[Tul96]   D.M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *The 22nd Annual Computer Measurement Group Conference*, San Diego, CA, December 1996.

[WB96]   S. Wallace and N. Bagherzadeh. A scalable register file architecture for dynamically scheduled processors. In *PACT-5*, pages 179–184, Boston, MA, October 1996.

[WG94]   D. L. Weaver and T. Germond. *"The SPARC Architecture Manual/Version 9"*. Prentice Hall, February 1994.

[Yea96]   K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[ZK98]   V. Zyuban and P. Kogge. The energy complexity of register files. In *ISLPED'98*, pages 305–310, Monterey, CA, August 1998.

[ZK01]   V. V. Zyuban and P. M. Kogge. Inherently lower-power high-performance superscalar architectures. *IEEE Trans. on Computers*, 50(3):268–285, March 2001.