CRIB: Consolidated Rename, Issue, and Bypass

by

Erika Gunadi

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Electrical Engineering)

at the

UNIVERSITY OF WISCONSIN-MADISON

2010

Abstract

This thesis is motivated by the growing problem of power bottleneck in high-performance CMOS processors. Recent and future nanometer CMOS technology processes do not provide as good voltage scaling, eliminating the quadratic effect in reducing power consumption across technology generation. Industry trend toward many-core processors also further limit the power available for each core.

Chip power distribution shows that a large percentage of chip power is spent on operand delivery. The power consumed by operand delivery is even larger than the power needed for the execution itself. Thus, we propose to reconstruct the way out-of-order execution is done in order to eliminate power consumption by operand delivery. We propose *CRIB: Consolidated Rename, Issue, and Bypass* as a solution to the power problem. Using CRIB, out-of-order execution is done without explicit register renaming. By removing explicit register renaming, several supporting structure needed for operand delivery can be eliminated. Hence, power consumption related to operand delivery can be dramatically reduced. CRIB also removes separation of data and dependency linking used in conventional out-of-order machine, resulting in the removal of speculative scheduling. With the removal of speculative scheduling, various cache optimizations that were not attractive for conventional machine due to the added latency non-determinism can be employed.

Our detailed energy evaluation indicates that the average energy saving in the execution core for SPEC CPU 2006 integer and floating-point are 72% and 67% respectively compares to a conventional out-of-order machine with a large instruction window. Adding cache optimizations further increases the energy saving to 80% and 77% respectively. Assuming that the front-end, clock-tree, and L2 cache consume 50% of total core power

and that the energy consumption for them remains constant, the energy saving translates into 40% and 38% of core energy consumption. This energy saving is gained without much performance degradation. Without cache optimization, CRIB's IPC is 12% lower than conventional out-of-order machine on average. However, with cache optimization, the average IPC is 1% higher and 8% lower than a conventional out-of-order machines for SPEC integer and SPEC floating point respectively, giving an average of only 3.5% slow-down. CRIB also has smaller area for the affected structures.

Acknowledgements

I would like express my deep gratitude to my advisor, Professor Mikko Lipasti for giving me a chance to work in his Pharm laboratory. The knowledge that I have gained while working there will be an indispensable asset in any of my future undertakings. I could not have made it this far without his unconditional support, availability, and eternal optimism. His critical thinking and deep understanding of computer architecture has helped me grow as a researcher and engineer, and I look forward to collaborating with him as a colleague.

I would like to thank my parents Hendrata Gunadi and Susi Hidajat. They helped me realize the fundamental importance of a solid education, and I am grateful for the freedom and resources they gave me to pursue my interests from an early age. I am especially thankful to them for raising me in a way that placed no preference or emphasis on gender. I also would like to thank my uncle Khoe Ging Hauw. As the first Doctor in our family, he inspired me to always reach for higher education and this aspiration led me to this school.

I would like to thank my husband Andrew for all the moments we spent together and for everything he has done for me, especially during the times when I was writing this work. He has been willing to cook for me during the most busy part of the writing. I especially thankful for his efforts in organizing our wedding while I was busy writing this thesis and preparing for the defense. I could not have done it without you. I look forward to starting the next chapter of our life together.

The University of Wisconsin computer engineering department and computer architecture group has provided a top-notch environment to develop my engineering and research skills, from its insightful and ambitious graduate students to its world-class fac-

ulty members. I would like to thank Professor Kewal Saluja for all his advises during my graduate school career. I would especially like to thank Ilhyun Kim and Shiliang Hu for the simulator infrastructures that I am using for this work, without which I could not have collected the data that appears in this thesis. I also would like to thank Mikko's prior Ph.D. students: Kevin Lepak, Trey Cain, Ilhyun Kim, Gordon Bell, Lixin Su, Eric Hill, and Natalie Jerger, as well as Mikko's current students: Dana Vantrease, Mitch Hayenga, Andy Nere, Vaishali Karanth, and Dave Palframan for the friendship and company in the lab.

Computer architecture research often requires large quantities and varieties of computer hardware, and administering these machines is rarely an easy task. Bruce Orchard has gone far beyond the call of duty by providing exceptional and timely support to our research group.

Finally, I thank my thesis committee members for their helpful feedback and interest on earlier versions of this research, and their general support throughout my time at UW. I also would like to thank Steve Kunkel as my IBM fellowship sponsor for my last year of Ph.D.

Table of Contents

Chapter 1: Introduction 1
1.1 Power Saving Techniques 3
1.2 Conventional Out of Order Design 5
1.3 CRIB: Consolidated Rename, Issue, and Bypass 8
1.4 Thesis Contributions 10
1.5 Thesis Organization 12
Chapter 2: Related Work 13
2.1 Register Renaming and Physical Register File 13
2.1.1 Register Renaming 13
2.1.2 Physical Register File 16
2.1.2.1 Relaxing Physical Register File Constraint 17
2.1.2.2 Reducing Physical Register File Power 22
2.2 Distributed Microarchitecture 23
2.3 Execution Array 25
2.4 Data Cache 25
2.5 Chapter Summary 31
Chapter 3: Experimental Methodology 33
3.1 Simulation Environment 33
3.2 Machine Model 35
3.3 Benchmarks 36
3.4 Physical Modeling 39
3.4.1 Register Alias Table (RAT) 40
3.4.2 Reservation Station 42
3.4.3 ALU and Bypass Network 43
3.4.4 Load and Store Queue 44
3.4.5 Reorder Buffer 45
3.4.6 Register File 45
3.4.7 Data Cache 46
3.5 Performance Debugging 47

3.6 Chapter Summary 48
Chapter 4: CRIB: Consolidate Rename, Issue, Bypass. 51
4.1 CRIB Concept 51
4.2 CRIB as a Realistic Execution Core 56
4.2.1 Partitioned-CRIB 56
4.2.2 Memory Instructions 60
4.2.3 Branch Instructions. 64
4.2.4 Complex Integer Instructions 65
4.2.5 Floating Point Instructions 65
4.2.6 Miscellanous Details 66
4.3 CRIB with Deeper Pipeline 68
4.4 CRIB with Data Cache Optimization 71
4.4.1 Cache Banking 72
4.4.2 Cache with Line Buffers. 72
4.5 Chapter Summary 75
Chapter 5: CRIB Design 77
5.1 CRIB Entry 77
5.1.1 CRIB. 82
5.1.2 CRIB with Deep Pipelining 86
5.2 Architected Register File 87
5.3 Modeling Results 88
5.4 Cache Optimization 90
5.4.1 Cache Banking 90
5.4.2 Line Buffers 91
5.5 Chapter Summary 93
Chapter 6: Experimental Evaluation of CRIB 95
6.1 Sensitivity Studies 96
6.1.1 Performance Sensitivity to the Number of CRIB Entries 96
6.1.2 Performance Sensitivity to Different CRIB Partitioning 99
6.1.3 CRIB Occupancy 99
6.2 Energy and Performance Comparison 102
6.2.1 Energy Comparison 102

6.2.2 Performance Comparison . . . 105
6.3 Cache Optimization . . . 108
6.3.1 Energy Saving . . . 108
6.3.2 Performance Effect . . . 110
6.4 Detail Benchmark Analysis . . . 111
6.4.1 Bzip2 . . . 111
6.4.2 Libquantum . . . 112
6.4.3 Mcf . . . 113
6.4.4 Omnetpp . . . 114
6.4.5 Perlbench . . . 115
6.4.6 cactusADM . . . 116
6.4.7 dealII . . . 117
6.4.8 GemsFDTD . . . 118
6.4.9 leslie3d . . . 119
6.4.10 povray . . . 120
6.4.11 Benchmarks Analysis Summary . . . 121
6.5 Chapter Summary . . . 122
Chapter 7: Conclusion . . . 123
7.1 Future Work . . . 126
7.1.1 CRIB's Window Size . . . 126
7.1.2 Front-end Energy Reduction . . . 129
7.1.3 Fission and Fusion . . . 129
7.1.4 Software Aspect . . . 129
7.1.5 Impact on Design Methodology and Flow . . . 130
7.2 Chapter Summary . . . 131
References . . . 133

List of Figures

FIGURE 1-1: Minimum Operating Voltage Trend 2
FIGURE 1-2: Data Movement Comparison. 6
FIGURE 1-3: Chip Power Breakdown 7
FIGURE 1-4: CRIB Data Movement Comparison. 8
FIGURE 1-5: CRIB Pipeline Diagram 9
FIGURE 3-1: Pipeline Diagram 35
FIGURE 3-2: Percentage of Committed Uops 39
FIGURE 3-3: Register Alias Table 40
FIGURE 3-4: Reservation Station. 42
FIGURE 3-5: ALU and Bypass Network 43
FIGURE 3-6: Load / Store Queue. 44
FIGURE 3-7: Data Cache 46
FIGURE 4-1: CRIB Concept. 52
FIGURE 4-2: CRIB Example. 54
FIGURE 4-3: CRIB Pipeline Diagram 55
FIGURE 4-4: Four-Partition CRIB 56
FIGURE 4-5: Completion Bit Propagation in Partitioned CRIB. 58
FIGURE 4-6: Producer-Consumer Pipeline Diagram. 59
FIGURE 4-7: CRIB Load Instruction Handling. 60
FIGURE 4-8: Recovery Example on Load-Store Misorder 62
FIGURE 4-9: CRIB with additional checkpointing capability. 63
FIGURE 4-10: CRIB branch misprediction recovery. 64
FIGURE 4-11: Floating Point CRIB. 66
FIGURE 4-12: CRIB with Deeper Pipeline 68
FIGURE 4-13: CRIB Pipeline Diagram. 70
FIGURE 4-14: Execution Schedule Diagram. 71
FIGURE 4-15: Cache Organization with Line Buffers. 73
FIGURE 4-16: Line Buffer Operation. 74
FIGURE 5-1: CRIB Entry. 78
FIGURE 5-2: CRIB Entry with Deep Pipelining 79
FIGURE 5-3: Output Router with Glitch Avoidance. 80

FIGURE 5-4: Proposed CRIB Partition Floor Plan 85
FIGURE 6-1: Number of Entries Sensitivity Study. 97
FIGURE 6-2: Partition Configuration Sensitivity Study. 98
FIGURE 6-3: CRIB Occupancy. 100
FIGURE 6-4: Energy Comparison. 101
FIGURE 6-5: Dispatched Instructions. 103
FIGURE 6-6: Total Pipeline Flushes. 104
FIGURE 6-7: Normalized IPC Comparison. 105
FIGURE 6-8: Energy Comparison for Cache Optimization. 107
FIGURE 6-9: Hit Rate of Line Buffers. 108
FIGURE 6-10: IPC Comparison for Cache Optimization. 109
FIGURE 6-11: Bzip2 Loop. 111
FIGURE 6-12: Libquantum Loop. 113
FIGURE 6-13: Omnetpp Loop. 114
FIGURE 6-14: Perlbench Loop. 115
FIGURE 6-15: cactusADM snippet. 116
FIGURE 6-16: dealII Loop. 118
FIGURE 6-17: GemsFDTD Code Snippet. 119
FIGURE 6-18: Leslie3d Loop 120
FIGURE 6-19: povray snippet. 120
FIGURE 7-1: Disjoint CRIB Diagram. 127
FIGURE 7-2: Virtual CRIB Diagram. 128

List of Tables

TABLE 3-1:Machine Configuration. 36
TABLE 3-2:Benchmark Descriptions 37
TABLE 3-3:Benchmark Descriptions and Baseline IPC 38
TABLE 5-1:Delay, Energy, and Area for CRIB 83
TABLE 5-2:Delay Characteristic of Regular and Transparent Flip-Flop 88
TABLE 5-3:Energy and Area of Architected Register File in one CRIB partition 88
TABLE 5-4:Energy and Area of Baseline Machine. 89
TABLE 5-5: Banking Results Comparison for 64KB, 64B, Direct Mapped Cache 90
TABLE 5-6:Delay and Energy Components for 64KB, 64B, DM, 4Bank Data Cache with Line Buffer. 91
TABLE 6-1: Program Characteristic Impact on CRIB. 121

Chapter 1

Introduction

Computer architects face broad new challenges in sub-50nm CMOS technology. Historically, power consumption in high-performance CMOS processors was only a secondary design constraints because supply voltage scaling across technology generation had a quadratic effect that held power in check, even as frequency and device density increased. Unfortunately, recent and future nanometer CMOS technology processes no longer provide similar voltage scaling, leading to serious power bottleneck for aggressive deeply-pipelined processors. Figure 1-1 shows the minimum operating voltage trend over the year across generations of various process technology. While the process scaling continues, minimum operating voltage does not scale much from 90nm technology forward, lingering around 0.8 Volts. One of the reason why V_{dd} has been scaling slowly starting from 90nm forward is leakage current control [41]. Leakage current is influenced by the transistor's treshold voltage, V_{th}. A decrease in V_{th} will result in exponential increase in leakage current. With V_{th} fixed, changing V_{dd} simply trades off energy and performance. The result is that from the 90nm technology forward, V_{dd} has been scaling slowly, if at all. Thus, we can no longer rely on process scaling to give us quadratic reduction of power. While device density continues to increase, thermal, power delivery, and energy supply constraints will severely complicate any effort to extract additional performance and functionality from an abundant supply of on-chip transistors.

The industry trend toward many-core processors further exacerbates the power budget problem. While the number cores in a chip keeps increasing, the chip power bud-

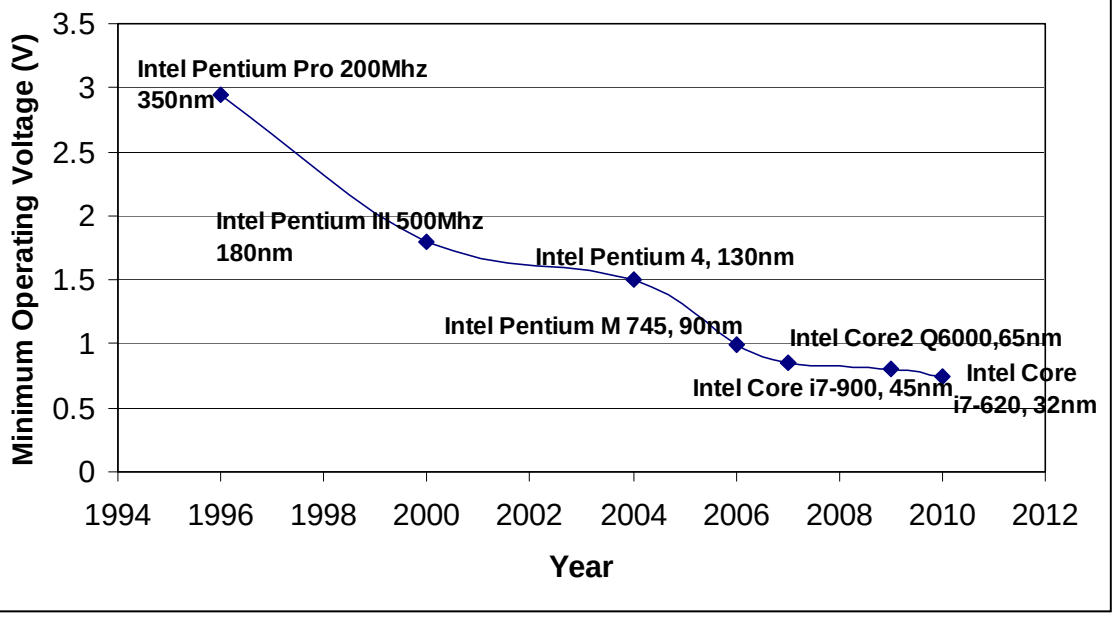


FIGURE 1-1. Minimum Operating Voltage Trend.

get does not increase as fast if at all; which further limits the power budget available for each core. Power density is also a big issue since more cores on a chip means more power dissipation. While the process scaling is able to shrink the core area and to keep the area the same despite the increasing number of cores, it is less successful in reducing the power consumption per core. Thus power density and heat dissipation in multi-cores environment also pose a big problem.

The objective of this thesis is to reduce core power consumption without sacrificing too much performance. It is successful in this regard, obtaining significant core power reduction while maintaining comparable performance with a conventional out-of-order processor. We provide details regarding our proposed implementation of *CRIB: Consolidated, Issue, Rename, and Bypass* in this thesis. Power saving techniques have been proposed abundantly in the past, both in academia and in industry. We summarizes those techniques and elaborates their weaknesses in Section 1.1. Because we propose to recon-

struct the way out-of-order machines are designed, Section 1.2 takes a deeper look on conventional out-of-order design and its problem. We summarizes the proposed technique in Section 1.3. Last we present the research contributions made by this work in Section 1.4.

1.1 Power Saving Techniques

Architecture level power saving technique proposed in the past can largely be categorized into microarchitecture level and system level. Microarchitecture level power saving approaches usually focus on certain structures and try to reduce the power consumption of the particular structure. However they usually have a side effect of reducing performance. While these approaches are usually able to reduce significant power consumption on the particular structure they focus on, the power saving is diluted enough in the chip level power consumption that it is not worth the performance degradation or the added complexity. For example, a technique claimed to save 30% of instruction scheduler power while having 5% performance impact would only save roughly 2%-3% of chip power. Thus, its chip power saving is less than the performance impact it introduces. Even though there have been many proposals in the past, they are rarely adopted by industry due to the problem with power saving dilution in the chip level.

Power saving proposals focused on reservation station include instruction packing [73], scalable low power issue queue [87], and SEED [60]. Proposals focused on the physical register file include register file port reduction [68][49][10], banked register file [86], and narrow operand exploitation [53]. Work focused on data cache and load store queues include cache way prediction [42][71][90], cache buffering [79][80][52][27], and load store queues improvement [11][32]. These prior work is discussed further in Chapter 2.

System level power saving approaches usually rely on voltage and frequency scaling. While it still reduces the performance, the power saved is usually larger due to the fact that the technique is applied to the whole cores, rather than just to a specific structure. It is usually the preferred power saving method by industry as the power performance trade-off is very good. Voltage frequency scaling is able to provide up to cubic power saving relative to performance impact, i.e. 5% performance reduction could potentially save 15% of power. However as we explained earlier, the minimum operating voltage has not been able to scale down as much. Thus the margin for power reduction using dynamic voltage frequency scaling is also getting more and more limited.

Industry have also started to use other approaches to cope with power budget problem. One approach is to have low power in-order cores rather power hungry out-of-order cores as used in Sun Niagara [54] and Intel Atom [26]. While certain application domains can tolerate low-power in-order cores by compensating for their low single-thread performance with abundant thread-level parallelism, general-purpose systems still require the single-thread performance and high instruction-level parallelism provided by aggressive out-of-order processors. For this class of systems, an ideal many-core processor core would provide the seemingly contradictory attributes of modest area, low power consumption, high instruction-level parallelism, and competitive frequency.

In this thesis, we propose to fully reconstruct the way conventional out-of-order machines are designed. Even though our approach is a microarchitectural approach, we do not tackle one specific structure. Instead, we introduce a different way of doing out-of-order execution. Thus our approach affects most structures in the execution core, resulting in significant power saving even in the chip level. In order to understand the weakness of

conventional out-of-order design that has to be redesigned, we take a deeper look on the design principle of conventional out-of-order machine in Section 1.2.

1.2 Conventional Out of Order Design

Conventional out-of-order designs have their origins in the earlier performance-focused and device-poor era. At the time, power was not a problem and transistors were expensive. These designs focused on maximizing utilization and reducing area, resulting in a minimal number of heavily pipelined functional units. Register renaming was employed to expose the underlying data flow graph's inherent parallelism, while clever scheduling algorithms efficiently time-multiplexed instruction execution on the precious functional unit pipelines.

Register renaming is a technique to maintain true dependencies while relaxing false dependencies. It allows independent instructions previously constrained by false dependencies to execute in an out-of-order manner while enabling true dependent instructions to get executed in a back-to-back manner. While successful at extracting instruction level parallelism and delivering excellent performance, conventional register renaming requires various structures for operand delivery such as the register alias table, the reservation station, the payload RAM, the register file, and the reorder buffer. These structures are constantly evaluating every cycle, resulting in significant power consumption. Tags and data are moved from one structure to another while status bits are updated in different places every cycle. Power is wasted to set up crossbars and port connections from one place to another and to latch status and data between pipeline stages. Figure 1-2 shows the diagram of structures needed to support operand delivery in the two most common register

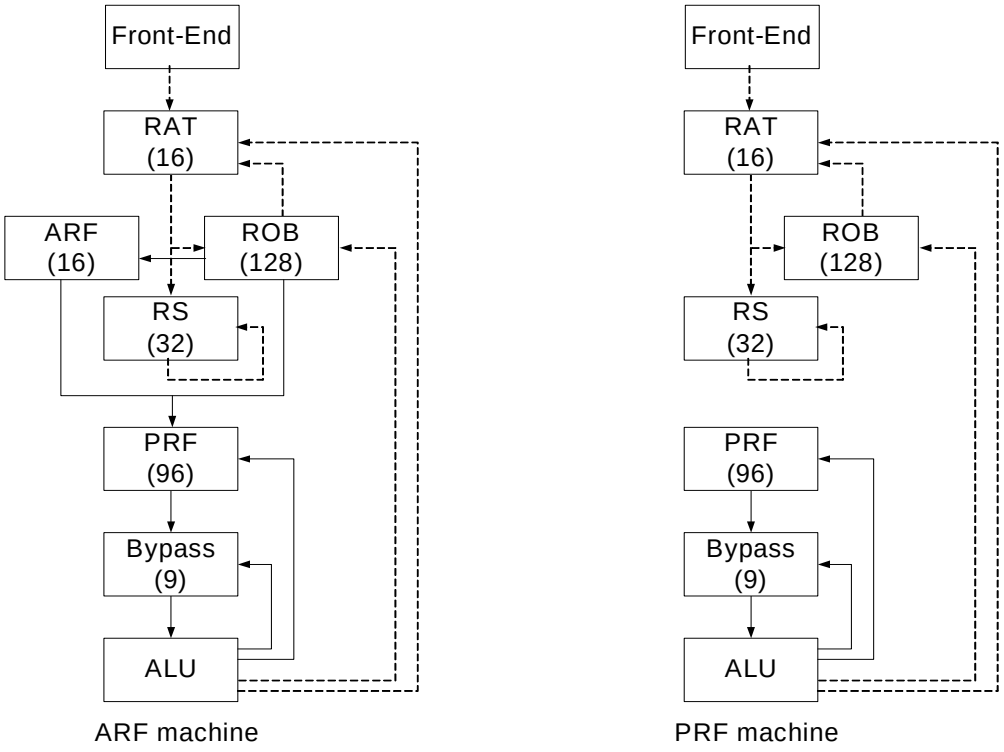


FIGURE 1-2. Data Movement Comparison. Solid lines represent data movement and dashed lines represent tag movement. The diagram is augmented with typical number of entries for each structure

renaming approaches: architected register file (ARF) and physical register file (PRF) machines. Dashed lines represent tag movement while solid lines represent data movement. The diagram is also augmented with typical number of entries for each of the structure. It can be seen from the diagram that in its lifetime, an instruction need to move data and tag multiple times on multiple structures, both in architected register file machine and physical register file machine. To make it worse, some of these structures, such as the reservation station, the payload RAM, and the bypass network, require associative search, which often requires significant power consumption.

It is widely known that the actual power required to do the actual computation is only approximately 10% - 15% of total chip power consumption. Figure 1-3 shows the chip power breakdown for the Intel Pentium Pro [14]. As shown in Figure 1-3, power con-

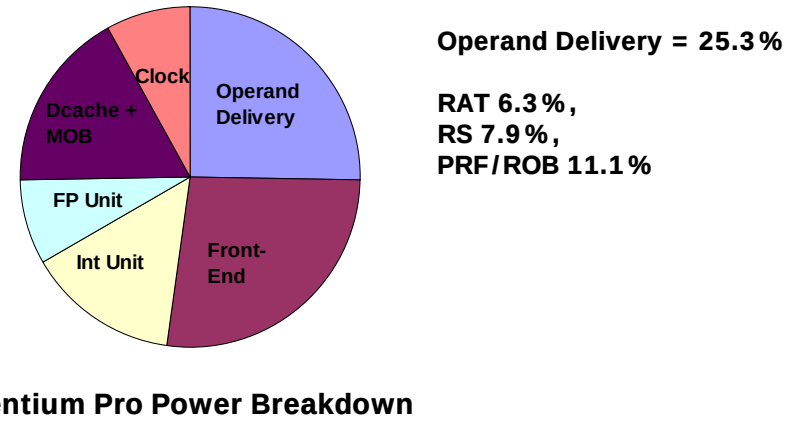


FIGURE 1-3. Chip Power Breakdown.

sumption associated with the operand delivery can be as large as 25.3 % of the total chip power. The 25.3 % of power consumption includes the power consumed by the register alias table (6.3%), the reservation station (7.9%), and the reorder buffer (11.1%). While the chip power breakdown in Figure 1-3 represents an older machine, we believe that operand delivery structures in current generation machine consumes similar amount of power if not more. To extract more instruction-level parallelism, current generation machines such as Intel's Nehalem have larger structures to support operand delivery, i.e. Intel Nehalem has an 128-instruction window while Intel Pentium Pro has a 40-instruction window. Thus the power consumed by these structures in newer machines should be at least the same if not more than the percentage shown in Figure 1-3.

This thesis focuses on eliminating the redundancy introduced by these operand delivery structures. From Figure 1-3, it can easily be seen that eliminating operand delivery structures would save 25% of chip power, which is quite significant. In a more aggressive out-of-order machine that has larger operand delivery structures, eliminating these operand delivery structures would save even more power.

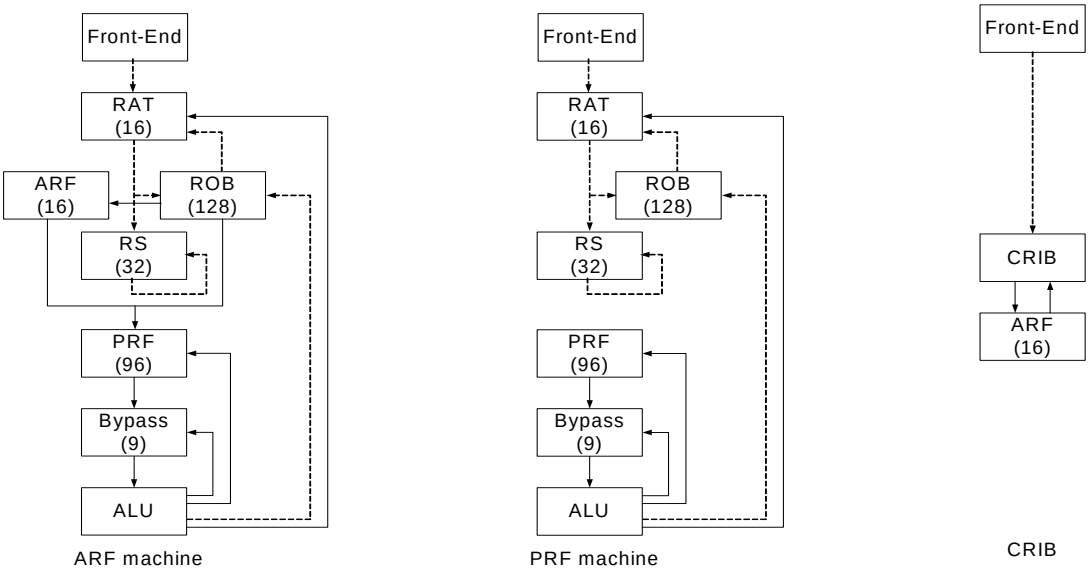


FIGURE 1-4. CRIB Data Movement Comparison. Solid lines represent data movement and dashed lines represent tag movement.

1.3 CRIB: Consolidated Rename, Issue, and Bypass

The primary goal of this thesis is to develop an alternative way to do out-of-order execution where explicit register renaming is no longer necessary. By removing explicit register renaming, supporting structures such as the register alias table, the reservation station, and the reorder buffer can be eliminated. Eliminating those structures will result in significant power reduction as those structures consume roughly 25% of total chip power as shown in Figure 1-3. The functionality of those structures are consolidated into one structure, the *CRIB*. Explicit register renaming is replaced with spatial renaming within the CRIB. A large multiported physical register file is replaced by a small spatially-organized architected register file which consists of a simple rank of latches. Data forwarding between instructions in the CRIB is accomplished without latches while keeping the design fully synchronous by latching only the control bits. This work represents a significant departure from prior microarchitectural power saving techniques that merely focused

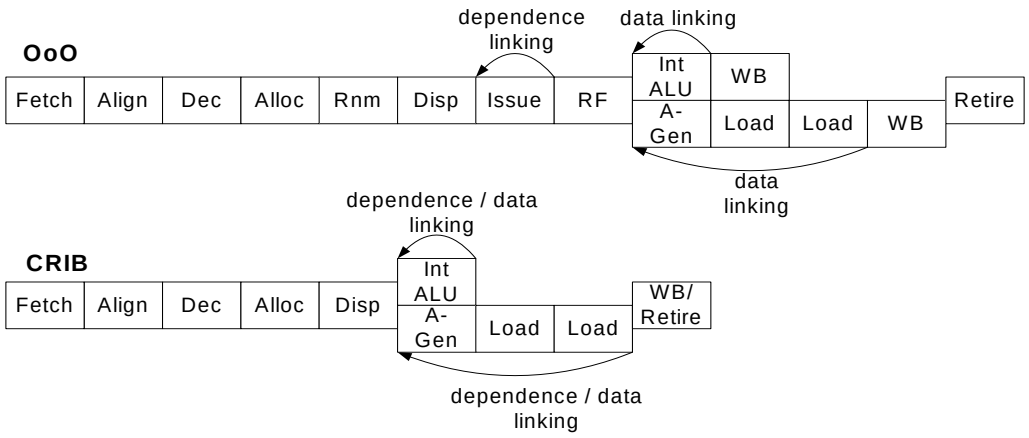


FIGURE 1-5. CRIB Pipeline Diagram.

on one or a few structures in the out-of-order core.

Figure 1-4 shows the tag and data movement comparison between architected register file style machine, physical register style machine, and CRIB. As shown, CRIB has a much cleaner tag and data movement compared to the conventional out-of-order design. Decoded instructions are inserted into CRIB from the front-end. The source tags are used to grab source operands from the architected register file. The destination tag is used to overwrite the destination register of the instruction. Once all instructions finish execution, the results are latched back into the architected register file.

By consolidating many of the operand delivery structures, CRIB is also able to remove several pipeline stages as shown in Figure 1-5. CRIB removes one pipeline stage at the front-end because of the removal of explicit register renaming. At the out-of-order window, CRIB removes issue stage and register file read stage and consolidate them in one stage only, the execution stage.

Different from conventional out-of-order processors where dependence linking is done separately from data linking, as illustrated in Figure 1-5, CRIB does dependence

linking and data linking together. A conventional out-of-order processor relies on speculative scheduling where dependent instructions are scheduled before the result to be consumed is ready. This speculative scheduling technique is vulnerable to non-determinism in execution latency, such as in data cache latency. The most commonly used latency, hit latency, is assumed to wake-up the dependent instructions. Upon a latency misspeculation, the dependent instructions are replayed. These replays increase energy consumption and cause performance degradation. The speculative scheduling has made many cache improvement techniques unattractive due to additional non-determinism in latency that these techniques often introduced.

CRIB, on the other hand, does not separate dependence and data linking. In other word, CRIB does not rely on speculative scheduling to wake-up dependent instructions. Thus, CRIB is able to use cache improvement techniques without much effect. In this thesis, we pair CRIB with cache-banking for additional bandwidth. Because CRIB mostly reduce power consumption in the execution core, we also revisit and employ cache line buffering [79][80][27] to further reduce power in data cache.

1.4 Thesis Contributions

The research presented in this thesis makes the following contributions:

- **Elimination of explicit register renaming:** We propose a mechanism to allow out-of-order execution without explicit register renaming. Instead, spatial renaming is used to relax false dependencies and enforce true dependencies.
- **Consolidation of operand delivery structures:** We propose to consolidate operand delivery structures such as the reservation station, the reorder buffer, and the

bypass network into one structure, the CRIB.

- **Aggressive reordering of store-load instructions and selective re-execution of mis-ordered load instructions:** Conventional out-of-order processors use complex predictor to predict if a load instruction can be ordered aggressively with respect to earlier stores. Mis-speculated load instructions trigger a flush for recovery, while false positive predictions will hinder the performance improvement that potentially be gained using this predictor. CRIB allows selective re-execution of mis-ordered load instructions because of the nature of its data dependency linking. Because the recovery mechanism is very light weight, CRIB can do aggressive reordering without the help of complex predictor. This aggressive reordering also helps to avoid false positive predictions that inhibit reordering unnecessarily.
- **Light-weight maintenance of branch misprediction recovery and precise exceptions:** Branch misprediction recovery and precise exception handling require complex mechanisms in conventional out-of-order machines, ranging from checkpointing mechanism, retirement rename table, and reorder buffer unrolling. In CRIB, branch misprediction and precise exceptions can be handled using a single signal to nullify all the younger instructions. The positional nature of the CRIB will then produce the precise architecture state after nullifying all the younger instructions.
- **Development of control pipelining:** We develop a mechanism to remove latches from the datapath. The design is kept fully synchronous by pipelining the control bits only. Because the control path is fairly simple with much less latency in comparison to the datapath, deep pipelining can be applied aggressively without suffer-

ing the overhead of regular pipelining.

- **Consolidating data and dependency linking:** Unlike conventional out-of-order design, data and dependency linking is not separate in CRIB. This notion allows CRIB to employ cache improvement techniques that were deemed unattractive for conventional out-of-order design.

1.5 Thesis Organization

This thesis is organized as follows: Chapter 2 presents a detailed overview of prior work on register renaming and physical register file improvement techniques, distributed architectures, and cache power saving techniques. Chapter 3 discusses the experimental methodology used to collect the data that appears in this thesis, both architectural modeling for cycle-level performance / energy analysis and physical modeling for access latency, access energy, and area analysis. Chapter 4 explains the concept of CRIB in detail and how the CRIB is extended into a fully-functional core. Chapter 5 describes the design and implementation of CRIB. A detailed experimental evaluation of CRIB is presented in Chapter 6. Finally, Chapter 7 concludes this thesis and present additional avenues for future research.

Chapter 2

Related Work

This chapter is divided into five sections. Section 2.1 addresses various proposals on register renaming and physical register files. Section 2.2 considers proposals that remove large centralized structures and replace them with smaller distributed units. Section 2.3 looks at related work using an array of execution unit to help speed-up instructions processing. Section 2.4 describes various proposals for improving dynamic energy consumption and access latency on level-1 data cache.

2.1 Register Renaming and Physical Register File

Register renaming and the physical register file work together in enabling out-of-order execution. As the heart of out-of-order execution, it is not surprising that register renaming receives much attention from computer architects. Early work on register renaming focuses more on the register renaming algorithm itself since the register renaming algorithm has not fully matured, while more recent work focuses on making the physical register file more efficient.

2.1.1 Register Renaming

Register renaming algorithm was first introduced by Robert Tomasulo from IBM in 1967, known widely as the Tomasulo algorithm [84]. It was developed to preserve precedence while encouraging concurrency by allowing instruction issue past false dependences. Tomasulo algorithm utilizes a common data bus to broadcast results to multiple structures and a simple tagging scheme to differentiate instances of the same register.

When an instruction is issued, a tag is written to the register file corresponding to the location producing the result, i.e. the reservation station entry. Anytime a result is broadcasted via the common data bus, each register file entry compares its tag with the broadcasted tag. An update is performed on a matching comparison.

Out-of-order execution as introduced by Tomasulo creates a new problem, which is precise interrupts. Later on, Sohi and Vajapeyam [77] introduced the Register Update Unit (RUU). In this work, RUU is used as a mechanism to solve dependency resolution and to guarantee a precise state of the machine at the same time. In RUU, write-after-read and write-after-write dependencies are resolved by assigning a version number to different instances of the register, allowing independent instructions to be issued in an out-of-order fashion. RUU can be thought of as a combination of a reservation station and a reorder buffer. It is used to store all instructions dispatched into the window. Architected register tags, together with counters / version numbers, are used to resolve dependencies and to represent the latest instance of the register. The RUU monitors the result bus and register update bus to get a copy of the non-ready operands. As an instruction becomes ready, it is sent to the functional units. The result is sent back to the RUU via the result bus. As an instruction commits, its result is sent to the register file via the register update bus. Because the register file update is done in an in-order manner, RUU is able to guaratee a precise state of the machine.

Another variation of register renaming is the block-structured ISA proposed by Melvin and Patt [59]. Similar with [77] and [85], the main motivation of the work is to eliminate write after read and write after write aliases and to exploit instruction level parallelism. Block-structured ISA introduces a new ISA that clusters instructions into an

atomic basic block with help from the compiler. Logical registers are only used to communicate inter-basic block. Inside the basic block, register tags are renamed using temporary physical register identifiers, allowing independent instructions previously limited by false logical register dependencies to be issued in any order. Once the write-after-read and write-after-write aliases are removed, the instructions are scheduled based on true data-flow dependences. This work is similar in spirit with current register renaming approaches. However, current register renaming approaches rename all logical register identifiers into physical registers, not only the ones inside a basic block. The differences are that 1) conventional renaming is done using a rename table rather than by the compiler, 2) valid logical register view is maintained per instructions boundary rather than per basic block boundary, 3) checkpointing, a retirement rename table, or ROB rollback is used to reconstruct valid logical register view on branch misprediction.

Prior register renaming work that is most similar to our approach is the Ultrascalar by Henry et al. [36][37]. While similar in concept, Ultrascalar and CRIB have different objectives that lead to differences in realizing the concept. The Ultrascalar objective is to create an extremely wide-issue machine, while CRIB objective is power saving.

Ultrascalar is motivated by the fact that simply increasing issue width in today's microprocessors result in a quadratic increase in the delays through many circuits, thus making future scaling of today's designs problematic. Ultrascalar proposes an alternative microarchitecture that scales well with issue width and window size. The Ultrascalar uses a circuit-switched network to compile at runtime the dataflow graph for a dynamic sequence of unrenamed instructions. The approach of removing explicit register renaming and creating a dataflow graph at runtime from unrenamed instructions is similar to the

CRIB concept. The unrenamed instructions are placed in a datapath, in which the producer instructions are connected to the consumer instructions in a ring-like structure. Similar to CRIB, the functional unit is also replicated for every instruction. However, Ultrascalar's datapath is fully synchronous. Execution results are latched on every unit. On the other hand, CRIB removes latches from its datapath, relying simply on the synchronous control path.

Another main difference between Ultrascalar and CRIB is the way the design is scaled up. Due to the fact that UltraScalar always latches its register value, it takes *n*-cycles for an execution result to travel from producer instruction to the consumer instruction if they are separated by *n* instructions. To reduce the delay, UltraScalar uses a complex hierarchical switch network to scale up its datapath. With the switch network, the delay to pass data from producer to consumer is now increasing logarithmically manner rather than linearly. On the other hand, CRIB maintains the simple ring approach and uses grouping of execution unit into a CRIB partition and a deep-pipelining approach to cut down the dataflow latency.

2.1.2 Physical Register File

In the vein of making the register file allocation and deallocation more efficient, a vast amount of prior work has also been done. These differ from our approach, which eliminates the physical register file completely due to the removal of renaming. These related works focus more on a better renaming and allocation algorithm. One objective of these works is to relax the physical register file constraint since the physical register file is considered as one of the design bottlenecks in current microprocessors. The other common objective is to save physical register file power.

Physical register file is one of the critical delay paths in a modern superscalar processor. The demand for higher performance drives computer designers to deeper pipelines, wider superscalar issues, and larger out-of-order windows to extract maximal instruction-level parallelism. Simultaneous multithreading (SMT) that is commonly used to maximize throughput also increases the demand for a larger physical register file. All these objectives exacerbate the design of the physical register file, since the minimum number of physical registers required is determined by the number of threads and the number of additional registers for inflight instructions. Farkas et al. [25] investigated the register file requirements of dynamically scheduled processors using register renaming with the SPEC92 benchmarks. Its finding is that roughly eighty registers are needed for a four-issue machine with 32-entry issue queue. For an eight-issue machine with a 64-entry issue queue, 128 registers are necessary. Farkas et al. [25] also claims that the increase of the cycle time due to the increase of the physical register file can have a significant impact on the overall performance going from a four-issue to an eight-issue machine.

Moudgill et al. [63] proposed one of the earliest works in the more-efficient register renaming scheme. In a conventional register renaming scheme, the previous definition of an instruction's destination register is released at retire stage. Moudgill et al. [63] proposes a more aggressive register reclamation. A physical register can be reclaimed as long as the following conditions are true: 1) the value has been written, 2) all issued instructions that need the value have read it, 3) the physical register has been unmapped. The proposal uses complete bits to track the register write, reference counters to track the readers of the register, and unmap flags to know if a later instruction has remapped the register.

While enabling a more efficient usage of physical registers, the proposed scheme does not handle precise exceptions. It can happen that an instruction between the the defining instruction and the last reader of the register throws an exception or a branch misprediction while the physical register has been reclaimed and rewritten. In that case, the register value is already gone and can no longer be recovered. *Cherry* [58] and *checkpoint processing and recovery (CPR)* [7] combine the aggressive reclamation proposal [63] with a checkpointing architecture to enable early recycling of physical register while maintaining precise exceptions. Cherry [58] adds the following condition for early reclamation: 1) the defining instruction and all the consumers must be executed and free of replay traps and branch mispredictions, 2) the instruction that remaps / supersedes the register is not subject to branch mispredictions. Cherry adds a superseded bit and a pending count for every physical register. CPR [7] modifies the usage of reference counters in [63]. Rather than incrementing the counter as readers enters the rename stage, CPR increments the counter whenever a checkpoint is created. All physical registers belonging to the checkpoint should not be released until the corresponding checkpoint is released. Thus, the counters for all physical registers belonging to a checkpoint are incremented when a checkpoint is created and decremented when the particular checkpoint is released.

Checkpointing mechanisms [7][58] enable the usage of aggressive reclamation while maintaining precise state of the machine. However, these mechanisms still require re-execution of the correct path when a branch misprediction or exception happens not on the checkpoint boundary. To avoid re-execution, Gonzalez et al. [29] proposed *multi-state processor,* a different mechanism to provide precise recovery of execution while enabling the implementation of large-window processors with a large physical register file. Multi-

state processor divides the physical register file into as many banks as the number of logical registers. Each logical register can only be renamed into its own physical register bank. Within such a bank, physical registers are allocated and released in order. *State control table (SCT)* manages the state of each physical register. Each SCT has a rename pointer that points to the last allocated physical register and a release pointer that points at the first physical register in the bank that cannot be released. A physical register cannot be released if the value has not been produced or it has not been consumed by its reader. Each physical register also records the number / state id of the instruction assigning it. A global *last committed state (LCS)* computes the oldest state Id of all the release pointers in each SCT. Any stateId older than LCS can be committed. A change in LCS might commit multiple older states at once. When a branch misprediction or exception occurs, the processor state is reset to the stateId of the instruction. The recovery stateId is broadcast to all SCTs. All physical registers with stateId greater (younger) than the recovery stateId are released.

Lipasti et al. [56] proposed *Physical register inlining*, which exploits the fact that many register values are narrow-width and can be represented with only a small number of bits. Those register values are stored directly in the map table, and the physical registers allocated to them are released. A significance-checking logic that verifies whether or not all n high-order bits of computation results are either ‘1’ or ‘0’ is added to the retire stage. If the result is narrow, it is written to the rename table in the following stage. When a narrow value is written to the rename table, each checkpoint copy of the rename table contains a stale pointer that points to that particular register. An eager approach is to update all the checkpoint copy. Another way is to increase the reference counter to a particular register whenever a rename table checkpoint that points to the register is created. A register is

not allowed to be released when its reference counter is not zero. This physical register inlining is applied together with the earlier aggressive reclamation [63] proposal to produce further performance improvement.

Ergin et al. [24] proposed *Register packing* that exploits narrow-width register values further by packing multiple results into a single physical register. Register packing proposes two schemes. The first one is conservative and allocates a full-width register for a computed result. If the computed result turns out to be narrow, the result is reallocated to partitions within a common register, freeing up the full-width register. The second scheme is more aggressive; it allocates register partitions based on a prediction of the width of the result and reallocates register partitions when the actual result width is higher than what was predicted. If the actual width is narrower than what was predicted, allocated partitions are freed up.

Jourdan et al. [44] proposed to exploit the high level of temporal locality (result redundancy) in result values using *physical register reuse*. Physical register reuse reuses a physical location whenever it detects that an incoming result value matches a previous one. This checking is performed during register renaming and requires some *value-identify detection* hardware. By mapping several logical registers holding the same value to the same physical register, physical register reuse opens up the opportunity for *sharing* and *result reuse & dependency redirection*. In *sharing*, the high level of value redundancy is exploited to either reduce the physical register file size or to effectively enlarge the active instruction window. In *result reuse and dependency redirection*, 1) result generation is moved from the functional units to the register renamer, eliminating the processed instructions from the execution stream, 2) the responsibility of generating results is moved from

one instruction to an earlier on in the instruction stream, possibly allowing dependent instructions to be scheduled earlier.

While the prior work mentioned so far are proposing aggresive register reclamation by exploiting various observable facts, *virtual physical register* [28][62] proposed a late allocation of registers. Physical registers are allocated at the end of the execution stage, rather than at rename stage as conventional processors do. However, deadlock can occur when an older instruction executes and there are no more physical register to allocate. In this case, younger instructions are not able to deallocate physical registers as their retirement is blocked by the older instruction that cannot find a physical register to allocate. Virtual physical register [28] solves the situation by guaranteeing that some number of oldest instructions in the window will obtain a register when they reach the write-back stage. The allocation scheme assigns registers to the oldest instructions in the window that have a destination operand, while the rest of physical registers are allocated on-demand. A later version of virtual physical register [62] changes the allocation policy during deadlock situation by stealing the register allocated for the younger instruction and assigning it to the older one. If there are more than one younger instruction with a register already allocated, the youngest one will be chosen.

In an effort to continue execution far beyond a cache miss, *kilo-processor* [21] and *continual flow pipeline (CFP)* [78] proposes a technique that dissociates 1) register release from the instruction commit process and 2) register allocation from instruction renaming. When instructions dependent on a cache miss enter the *slice-data-buffer*, physical register numbers are used as virtual tags only. Instructions subsequently reacquire physical registers when they reenter the pipeline. Kilo processors also combines checkpointing, late

allocation, and early release mechanism to create a physical register file that can support thousands of in-flight instructions.

Oehmke et al. [66] proposed *virtual context architecture (VCA)* to give an illusion of a machine with thousands of registers. In VCA, the physical register file is treated as a cache of a much larger memory-mapped logical register space. An individual instruction needs only its source operands and its destination register to be present in the register file to execute. Inactive register values are automatically saved to memory as needed, and restored to the register file on demand. The rename stage is modified to trigger the movement of register values between the physical register file and the data cache by inserting a load or a store instruction. Renaming in VCA is extended into a two stage process. First, the source and destination register indices in the machine instruction are combined with the thread's context base pointer to generate memory addresses. The second stage of renaming maps the logical register memory addresses to physical registers. The rename table is modified to include tags like a cache because VCA maps registers from a large, sparse space. A miss in the rename table indicates that there is no physical register mapping of the desired logical register. A free physical register is then allocated and the register value must be read in from memory using a load instruction. If there is no free physical register, a physical register already allocated to a different logical register must be freed and the value is stored in the logical register's memory location.

2.1.2.2 Reducing Physical Register File Power

One way to reduce physical register file power consumption is by reducing register ports. Park et al. [68] proposed bypass hint to avoid unnecessary register file reads where the values can be bypassed. Register file write port reduction is also used by pairing regis-

ter file banking and decoupled rename. Decoupled rename is a technique to separate dependence and physical tagging of register operands, thus avoiding bank conflict during writeback. Kim et al. [49] proposed to use an auxilliary structure called *Delayed Write-back Queue (DWQ)* to cache operands that are recently produced from the functional units. The DWQ reduces the need for write ports as well as the peak need for read ports. Balkan et al. [10] proposed to avoid writeback for transient values, values that are delivered to their consumers via the bypass network. Reducing the number of ports using multiple interleaved banks is also proposed by Tseng et al. [86].

Kondo et al. [53] proposed *Bit-Partitioned Register File* that exploit narrow width operands. Register entries are partitioned into smaller width, resulting in more register entries with smaller width. Multiple entries are allocated for one operand if its effective bit-width is bigger than the bit-width of one entry. Otherwise only one partition is used.

2.2 Distributed Microarchitecture

In recent years, the concept of removing large centralized structures and replacing them with smaller distributed functional units has been explored in Raw, TRIPS, and Wavescalar. Raw [88][55] microprocessors are made up of simple tiles, each with a portion of the register set, a portion of the instruction and data cache, and one of the functional units. These tiles communicate via a pipelined two-dimensional interconnect mesh. Each tile contains a simple five-stage pipeline to maximize the clock rate and the number of tiles on the chip. The hardware is fully exposed to the compiler to construct the schedule and spatial placement for instructions. The main difference between Raw microprocecessors and our proposal is that Raw machines still depend on register renaming algorithm to resolve

data dependences. It compensates for the scalability issue of the register file by distributing it across many tiles, creating non-uniform register access latencies (NURA).

In a similar spirit, TRIPS [65][72] uses an array of ALUs, each with limited control, connected by a scalar operand network. Each array consists of instruction and operand buffer, a functional unit, and router for input to and output from the array. With support from the compiler, programs are divided into groups of instructions. Each group occupies one array. Most data communication is done directly from producer to a consumer without involving the register file, thus reducing the read and write bandwidth to the register file significantly. The physical locations of consumers are explicitly encoded within producer instructions, thus no broadcast is necessary. The producer simply delivers its result to the listed consumers. To build a data-flow like execution, support from the compiler is required. Besides not requiring compiler support, one major difference between TRIPS and CRIB is how the data is passed from a producer to a consumer. In TRIPS, a producer has to deliver its result to consumers. while in our scheme, a producer only needs to write to its own destination register path.

Similar with TRIPS, Balfour et al. [9] uses explicit operand forwarding to reduce the size of the register file on the ELM processor. Explicit operand forwarding in the tile-architecture lets software orchestrates the routing of operands through the forwarding network to avoid writing ephemeral values to the register file. By letting the software capture short-term reuse and locality close to the functional units, energy efficiency is improved by allowing a significant fraction of operands to be delivered from inexpensive operand registers that are integrated with the functional units. Operand register files are small, inexpensive register files that are integrated with function units in the execute stage of the

Wavescalar [81][82] goes further to build its decentralized data-flow processors by embedding processing elements (PE) into the cache, called the WaveCache. The WaveCache is a grid of PEs arranged into a cluster. Each PE contains logic to control instruction placement and execution, input and output queues for instructions operands, communication logic, and a functional unit. The goal of WaveScalar is to break the serialization point of the von Neumann model, namely the program counter, and to guarantee load-store ordering. By distributing the processing elements, WaveScalar also eliminates all the large hardware structures that make superscalars non-scalable.

2.3 Execution Array

Arrays of execution units are also used frequently in reconfigurable computing. This execution matrix can be used as a stand-alone datapath such as in RaPid [23] or embedded into a regular processor such as in Garp [35]. This reconfigurable hardware (FPGA) is used to exploit applications containing repetitive tasks such as encryption or Fourier transform. When embedded as a coprocessor, the microprocessor itself is slightly modified to be able to transfer work to the embedded hardware. Special purpose instructions are added to load new configuration of the FPGA and form a new execution matrix as necessary.

2.4 Data Cache

Data cache has been known as one of the main power consumers in a chip. Roughly 20% - 30% of chip power is consumed by data cache [34][14]. Many techniques

have been proposed in the past to help reduce data cache power consumption. In this section, various microarchitecture techniques that can be used reduce dynamic power of data cache are described. While some of the earlier proposals described do not focus on reducing power, those proposals can easily be repurposed as power saving techniques.

While there have been many proposals to reduce data cache power consumption, only a few are currently used in today's microprocessors. Those currently being used are techniques that do not require speculation such as subbanking [79][80] and bit-line segmentation [27]. Techniques introducing additional non-determinisn in load latencies are hardly used. The reason is that current generation microprocessors rely on speculative scheduling in scheduling load dependent instructions. The scheduler assume the most commonly encountered load latency and schedules the dependent instructions based on the assumed latency. Upon latency misspeculation, the speculatively issued load consumers and their descendants must be squashed from the pipelines and re-issued. Some of the proposals have proposed to use means other than effective address to enable early checking of the latency speculation [50][51]. However, their techniques do not remove the speculation completely and are only applicable to certain power saving techniques.

In contrast to conventional out-of-order processors, CRIB is tolerant to non-determinism in load latencies. Thus we proposed to extend the idea of line buffers [79][80][27] in our cache design and combine it with partial tag match [57][61][31] for fast verification. Rather than treating the line buffer as a separate entity as in prior work, the line buffer is embedded in the cache in our design. By embedding in the cache, it allows us to naturally exploit the notion of sub-array partitioning in today's cache design and having multiple line buffer. It also reduced the necessity to have large number of wires to fetch

the whole cache line (typically 64 bytes) out from the cache.

For the rest of this section, we will summarize different cache optimization techniques have been proposed in the past. Most of the optimization can be categorized into sequential access in set-associativity, cache partitioning, and buffering.

Sequential access in set-associativity cache has been one of the common cache optimization proposals. Agarwal et al. [4] proposed the Hash-Rehash cache to reduce the miss rate of direct-mapped cache to be similar to that of set associativity cache. Hash-Rehash cache starts with a direct-mapped cache and treats a pair of a set in a similar way as associativity, i.e. for 8-set cache, set 2 (010) and set 6 (110) are considered a pair. Cache access always starts from the set of the pair that has the same index as the address. On a miss, the next set in the pair is accessed. When a miss occurs, the newly fetched data is placed in an indexed set and the data currently in the indexed set is moved to the paired set. Agarwal et al. later proposed a rehash bit to further improve Hash-Rehash cache. The newer proposed cache is called Column-Associative Cache [6]. The rehash bit indicates that the other set contained rehashed data that will result in a miss. Thus, the second set does not have to be checked to further save miss latency. Both [4] and [6] relies on static information on which set in the pair to be accessed first.

In contrast with [4] and [6], Chang et al. [18] and Kessler et al. [46] used dynamic ordering to guide sequential probing in set-associativity cache. Chang et al. [18] proposed a cache organization where the cache provides the block selected by the MRU information upon access. Concurrently, all tags are compared; if the wrong cache block is used, it is detected and the proper block is provided in the next cycle. Kessler et al. [46] proposed a similar organization. Each cache block in a set uses a MRU bit to indicate the most

recently used block. The block indicated by the MRU bit is probed first. If the data is not found, the second block is probed. If the data is found, the MRU bit is inverted, indicating the second block is more recently used than the first block. If the data is not found, the least recently used block is replaced. While dynamic information is shown to be superior to static information, it requires a longer latency as the MRU information has to be fetched prior to accessing the cache contents to determine which block should be examined first.

Calder et al. [16] proposed predictive sequential associative cache (PSA-Cache) that combined the best of static and dynamic technique described above. It uses a MRU policy to guide the replacement policy while utilizing rehash bit from [6] to avoid unnecessary checking. In order to guide the selection of which block to probe first, PSA-Cache uses steering bits. Each line of the cache has each own steering bit, for example a 4-set 2-way associative cache has 8 steering bits. While using the effective address to index steering bits proved to give the best result, it incurred additional delay for steering bit access. Another variant to index the steering bits is using an XOR of register content (base address) and the address offset. This variant gives the next best result while having less latency.

In an effort to reduce cache energy, Inoue et al. [42] and Powell et al. [71] proposed to use way prediction that was originally proposed for reducing latency to design low power caches. Only the predicted way is accessed first and other ways are accessed on prediction miss. Inoue et al. [42] uses a simple MRU prediction to guide the way access while Powell et al. [71] uses the load PC to access a way predictor. Powell et al. [71] also proposed to use selective direct-mapping to further reduce cache energy. Cache sets are converted into direct-mapped in non-conflicting cases. Conflict prediction table is

Witchel et al. [90] proposed a direct-addressed cache, which uses a hardware-software design for an energy-efficient data cache. Direct addressing allows software to access cache data without a hardware cache tag check. These tag-unchecked loads and stores save the energy of a tag check when the compiler can guarantee that an access will be to the same line as an earlier access. Support for tag-unchecked loads and stores were added to C and Java compilers.

Gunadi and Lipasti [31] used partial tag match to help guide way selection in the data cache. The partial result is generated by the narrow width scheduler. The data cache is banked into narrow and wide banks. The partial result from the scheduler is used to access a narrow cache bank and generate way selection. The way selection is then used to access only the matched way in the wide-bank. If there is no match in the narrow bank, the access to the wide-bank is disabled to further save energy.

The optimizations described so far use the idea that a cache way can be accessed in a more serial manner in order to save some power. Another common optimization is fragmenting caches into smaller banks so that only a small subset of the cache has to be activated during access. This optimization is widely employed in today's caches. Temporal locality can also be exploited further by caching or buffering more recently used blocks.

Su and Despain [79][80] studied various optimizations for a low power cache. Optimizations being studied are block-buffering, cache-subbanking, and Gray code representation. A block buffer is another cache that is closer to the processor than the regular L1 cache. The block buffer contains the last line being accessed in the L1 cache. On a hit, the data is directly read from the block buffer and the cache is not operated. Cache-sub-

banking is a partitioning technique used in the cache so that only the necessary subset of block and rows are activated during an access. This technique is widely employed in today's caches as sub-array techniques. Lastly, Gray code is used to minimize switching activity in the data cache.

Kin et al. [52] proposed the filter cache, an unusually small cache placed in front of L1 cache. L1 cache is accessed only when there is a miss in the filter cache. Because the filter cache is smaller than the L1 cache, it will generally have a faster access time and less power consumption. Filter cache latency can be as small as one clock cycle in comparison to L1 latency that is usually 2-4 clock cycles.

Ghose and Kamble [27] used a detailed register-level simulator to study cache power saving techniques: subbanking, multiple line buffers, and bit-line segmentation. [27] extends the idea of a single line buffer [79][80] into multiple line buffers. These multiple line buffers represented the last four accesses into the cache. Set number comparison is done in parallel for all line buffers. Upon a match, corresponding tags are steered into the comparison logic. In subbanking, the wordline is segmented so that data line is now spread across a number of subbanks. Only the accessed subbank is activated. Bit-line segmentation is similar to subbanking, but is applied vertically to the bitline. This results in smaller precharge drivers and sense amplifiers. Similar to subbanking, only the segment being accessed need to be precharged.

Most proposals for low power caches introduce additional non-determinism in load latencies. In modern microprocessors, where speculative scheduling is widely used, additional non-determinism can result in wasted energy and performance loss. To alleviate the energy and performance impact of non-deterministic load latency, various techniques

have been proposed.

Kim et al. [50][51] proposed to bypass the address generation whenever the offset is detected to be zero in the decode stage. His study found that many displacement values are small numbers. Thus the displacement values often do not generate carry into the index bits. In this case, the base address from a register read can be used to directly bypass address generation. Thus, enabling the access to the cache or prediction table one cycle before the effective address is fully generated.

Another proposal to provide early information to reduce the non-determinisn introduced by the low-power cache design is narrow-width dynamic scheduling [31]. In the proposal, a narrow scheduler supplies partial operand values that is used to access narrow cache bank. The result from the narrow-bank access is then used to guide which way, if any, of the wide-bank cache need to be accessed.

2.5 Chapter Summary

Register renaming and physical register file is not at all a new research area. There are abundant prior works with various motivations ranging from performance improvement, scalability, and power savings. However, the main difference between prior work and CRIB is that most of the prior works merely propose a more efficient physical register file and register renaming techniques without completely deconstructing the way out-of-order machine is designed. Ultrascalar is the most similar prior work to CRIB. It has the same concept of removing register renaming altogether. However the difference in motivation has led into different realization of the concept.

CRIB can also be thought to be similar with distributed architecture work such as

Raw, TRIPS, ELM, and Wavescalar. However, the motivation itself is completely different. CRIB's motivation is power saving that leads to the removal of register renaming. On the other hand, the main motivation of the distributed architecture work is scalability. While physical register file modification to improve scalablility is a major part of prior work, none of them completely remove the register renaming. Instead, they may rely on the compiler to help with their modified renaming schemes.

Since CRIB has an array of ALUs, CRIB also bears similarity with prior work in specialized execution array such as Garp and Rapid. This prior work uses reconfigurable hardware that is used to speedup applications with repetitive tasks. In contrast, CRIB is used for general purpose computation without motivation of improving performance of such repetitive tasks.

Lastly, we pair cache optimization techniques with CRIB. Cache power saving techniques have been proposed many times in the past. Unfortunately, only a few have made it into industry. It is because most of these techniques introduce additional non-determinism in cache latency. Non-determinism in cache latency does not pair well with speculative scheduling commonly employed in today's microprocessors. In contrast, CRIB handles non-determinism well and pairs well with some cache optimization techniques such as cache banking and line buffers.

Chapter 3

Experimental Methodology

This chapter details the experimental methodology used to collect the data that appears in this thesis. It begins with a description of the simulation environment and relevant microarchitectural parameters of our baseline machine model. Next, we present information regarding the applications used to benchmark our implementation, followed by a brief discussion and data related to our baseline machine. Later, we explain physical modeling of latency, area, and power. Last, the performance debugging methodology is described.

3.1 Simulation Environment

All of the data presented in this dissertation was collected with an x86 architecture-level simulator derived from Bochs [2], an IA32 emulator. Bochs is used as the functional portion for the simulator. An internal RISC ISA is designed to crack x86 instructions into uops that are executed in the timing part of the simulator. The timing part of the simulator is loosely based on SimpleScalar [15]. The simulator is able to run the IA32 instruction set including x87, MMX, SSE, and SSE2 instructions. Activity counts are added to estimate energy consumption during the benchmark run, similar to Wattch [14].

The timing part of the simulator is able to simulate out-of-order execution. A rename-table is used to maintain the mapping between architected register identifiers and physical register identifiers. Rename-table checkpoints are used for fast recovery on

branch mispredictions. Inside the issue window, a full speculative scheduling and squashing recovery is performed. The simulator is also equipped with aggressive load-store reordering with pipeline flush as a recovery mechanism in the case of misprediction. A store-set predictor [20] is used for load-store reordering.

We implemented CRIB and its related hardware structures on top of this infrastructure. This required a number of substantial modifications and enhancement. One main difference is the way the dependencies are maintained and data is propagated between producers and consumers. The simulator assumes a conventional physical register file based out-of-order machine where the data is propagated through a centralized physical register file and bypass network. The physical register identifier is used to maintain dependencies inside the reservation station. Once an instruction is issued, its register identifier is used to wake up all dependent instructions inside the reservation station. On the other hand, CRIB removes the use of physical register identifier to detect dependencies between instructions. Instead, CRIB uses architected register identifiers and the position of instructions inside CRIBs to detect true dependencies. Thus, data and ready signal propagation in CRIB have to be done on an entry-per-entry basis according to how many entries can be traveled in one cycle. In each entry, a comparison has to be done to see if a write-after-write dependency occurs and the propagation of the previous data has to be stopped.

Another major enhancement is to enable deep-pipelining within CRIB. In deep-pipelining mode, the CRIB is clocked faster than the front-end of the machine; resulting in multiple clock domains in the machine. Nested loops are employed to run the deeply pipelined CRIB faster than the rest of the machine.

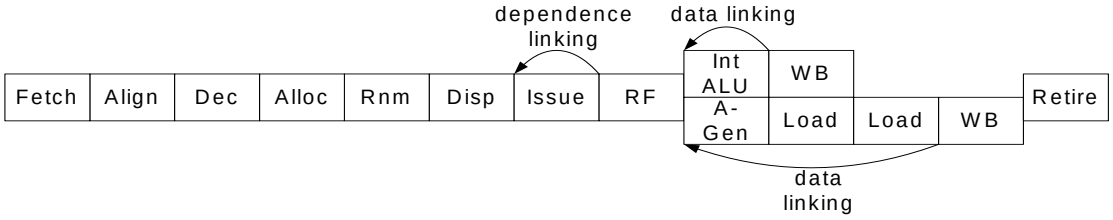


FIGURE 3-1. Pipeline Diagram. “

3.2 Machine Model

The x86 simulator being used supports a large number of configurable microarchitectural parameters that control the timing of the processor and its memory requests. Figure 3-1 shows the pipeline diagram of the baseline machine model that CRIB is compared against. Table 3-1 presents the rest of the machine configuration of the baseline machines. Two baseline machines are used. Baseline 1 is a small out-of-order core with a 24-entry ROB. Baseline 2 is a Nehalem-style machine with 128-entry ROB. Baseline 1 is configured to have a similar window size as our chosen CRIB model, while baseline 2 is configured to represent today’s microprocessor configurations. We also believe that baseline 2 will be a close match to future systems as well. Historically, higher transistor density enabled by decreasing feature sizes has lead to more aggressive processor cores in each successive technology generation. However, dramatic increases in power consumption have resulted in diminishing returns on single-thread performance, and have generated a trend toward designs that favor more cores over more complex cores. For this reason we contend that future microarchitectures will not be significantly wider, deeper, and more complex than they are today.

Table 3-1: Machine Configuration.

Attribute	Baseline 1 Small Out-of-Order Processor	Baseline 2 Nehalem-like Out-of-Order Processor
Pipeline Depth	11 stages	11 stages
Fetch Queue Size	16	16
Branch Predictor	combined bimodal (16k entry)/gshare (16k entry) with selector (16k entry); 32entry RAS; 2k entry 4-way BTB	combined bimodal (16k entry)/gshare (16k entry) with selector (16k entry); 32entry RAS; 2k entry 4-way BTB
Decode/Issue/Commit Width	4/4/4	4/6/4
Issue Window	16 with speculative scheduling and squashing recovery	36 with speculative scheduling and squashing recovery
Reorder Buffer Size	24	128
Physical Register File	32 Integer, 36 FP	96 Integer, 64 FP
Load/Store Queue Size	8/4	48/32
Memory Dependence Predictor	Store set dependence predictor with 4k ssit and 128 lfst, Flush recovery	Store set dependence predictor with 4k ssit and 128 lfst, Flush recovery
Functional Units (latency)	2 INT ALU (1) 1 INT MUL/DIV (4/4) 1 L1D memory ports (1+2) 1 ST data (1) 2 SIMD units (1) 2 FP ADD/MULT (6) 1 FP DIV/SQROOT (12)	3 INT ALU (1) 1 INT MUL/DIV (4/4) 1 L1D memory ports (1+2) 1 ST data (1) 2 SIMD units (1) 2 FP ADD/MULT (6) 2 FP DIV/SQROOT (12)
Prefetch	Stream prefetching on DL1 miss 32-entry prefetch buffer	Stream prefetching on DL1 miss 32-entry prefetch buffer
L1 Caches	I-Cache: 32KB, 2-way, 64B lines (2) D-Cache: 32KB, 4-way, 64B lines (2)	I-Cache: 32KB, 2-way, 64B lines (2) D-Cache: 32KB, 4-way, 64B lines (2)
L2 Cache	2 MB, 8-way unified, 128B lines (12)	2 MB, 8-way unified, 128B lines (12)
Memory	Off-chip memory: 168-cycle	Off-chip memory: 168-cycle

3.3 Benchmarks

The SPEC CPU2006 [3] Integer and Floating-Point benchmark suites provide a collection of standardized programs and inputs intended to represent real-world applications and programming constructs. We uses the result of redundancy analysis in [70] to choose eight benchmarks from SPEC INT 2006 and eight benchmarks from SPEC FP 2006 that are most representative of the whole suite. All benchmarks use the SPEC reference input sets. For benchmark having multiple input set, one input set that has program characteristics closest with the whole (aggregated) benchmark run is selected based on

.

Table 3-2: Benchmark Descriptions .

Benchmark	Language	Application Area	Description
SPECINT2006			
astar	C++	Path-finding Algorithms	Path finding library for 2D maps, including the well known A* algorithm
bzip2	C	Compression	Bzip2 version 1.0.3, modified to do most work in memory, rather than doing I/O
gcc	C	C Compiler	Based on gcc Version 3.2, generates code for Opteron
libquantum	C	Physics / Quantum Computing	Simulates a quantum computer, running Shor's polynomial-time factorization algorithm
mcf	C	Combinatorial Optimization	Uses a network simplex algorithm to schedule public transport
omnetpp	C++	Discrete Event Simulation	Uses the OMNet++ discreate event simulator to model a large Ethernet camput network
perlbench	C	Programming Language	Derived from Perl V5.8.7
xalancbmk	C++	XML Processing	A modified version of Xalan-C++, which transforms XML documents to other document types
SPECFP2006			
cactusADM	C, Fortran	Physics / General Relativity	Solves the Einstein evolution equations using a staggered-leapfrog numerical method
calculix	C, Fortran	Structural Mechanics	Finite element code for linear and nonlinear 3D structural applications. Uses the SPOOLES solver library.
dealII	C++	Finite Element Analysis	A C++ program library targeted at adaptive finite elements and error estimation
GemsFDTD	Fortran	Computational Electromagnetics	Solving Maxwell equations in 3D using the finite-difference time-domain (FDTD) method
lbm	C	Fluid Dynamics	Implements the Lattice-Boltzman Method to simulate incompressible fluids in 3D
leslie3d	Fortran	Fluid Dynamics	Computational Fluid Dynamics (CFD) using Large-Eddy Simulations with Linear-Eddy nModel in 3D
povray	C++	Image Ray-tracing	Image rendering
soplex	C++	Linear Programming	Solves a linear program using simplex algorithm and sparse linear algebra

clustering analysis [70]. Table 3-2 lists the SPEC CPU2006 benchmarks used and their descriptions.

All benchmarks were compiled with gcc version 3.3.2 with -O3 optimization. The pinpoint tool [69] is used to get simpoints [74][75] for each of these benchmarks. The benchmarks are fast-forwarded into the first simpoint before timing analysis is performed on the following 100 million instructions. During the fast-forward, branch prediction, level-1 instruction cache, and level-2 cache are warmed using the instruction stream.

Table 3-3: Benchmark Descriptions and Baseline IPC.

Benchmark	Input Set	FastFwd Instructions	IPC		EPI (pJ)	
			Baseline1	Baseline2	Baseline1	Baseline2
SPECINT2006						
astar	input set 2	200 M	0.679	0.780	123.310	219.315
bzip2	input set 4	3700 M	0.960	1.410	111.381	148.962
gcc	input set 1	16100 M	0.432	0.565	196.080	316.246
libquantum	input set 1	1300 M	2.118	3.000	70.799	84.056
mcf	input set 1	46600 M	0.170	0.176	352.960	834.315
omnetpp	input set 1	4500 M	1.681	1.998	76.188	93.140
perlbench	input set 1	500 M	0.804	1.127	120.274	173.517
xalancbmk	input set 1	95500 M	0.589	0.670	138.049	267.823
SPECFP2006						
cactusADM	input set 1	900 M	0.584	0.847	113.744	129.953
calculix	input set 1	400 M	1.356	1.603	80.584	103.154
dealII	input set 1	5900 M	0.933	1.080	88.372	102.312
GemsFDTD	input set 1	2900 M	0.370	0.443	102.882	115.674
lbm	input set 1	400 M	0.385	0.435	164.361	192.945
leslie3d	input set 1	400 M	0.362	0.448	100.382	116.619
povray	input set 1	62000 M	0.912	1.353	103.313	132.889
soplex	input set 1	18400 M	0.639	0.756	130.724	181.767

We use this benchmark set as our primary metric to evaluate CRIB. Table 3-3 indicates the input set being used and the number of fast-forwarded instructions before detailed timing analysis is performed. Table 3-3 also indicates their baseline performance in *instructions committed per cycle* (IPC) and baseline *energy per instruction* (EPI) in pJ. All performance and energy data in this thesis is normalized to the IPCs and EPIs of the baselines. This helps clarify the performance graphs and provides an easy way to display speedups and energy reduction.

Figure 3-2 shows the breakdown of committed uops during detailed timing analysis. Uops are categorized into different categories such as load, store, branch, integer, floating-point, and SSE instructions. SSE instructions here is defined as SSE instructions that do not fall into load, store, nor floating-point operations. The instruction breakdown is fairly uniform for SPEC INT benchmarks. It has roughly 20%-35% of load instructions, 10%-20% of store instructions, 15%-20% of branch instructions, and 40%-55% integer

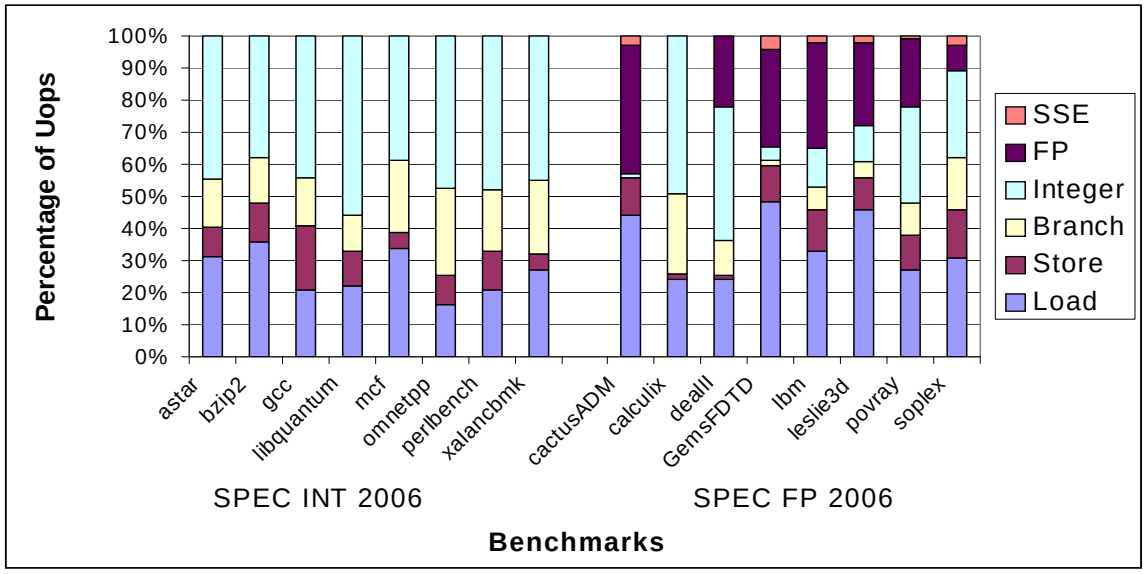


FIGURE 3-2. Percentage of Committed Uops.

operations. SPEC FP benchmarks shows higher load operations on average. The reason is that many of the floating-point instruction streams consist of a short dependency stream that is started with a load operation. Most of the FP benchmarks have at least 10% and up to 40% of floating point operations. However, calculix does not show any floating-point uops in its committed instructions because the first simpoint used have not reached the main loop of the program. A verification run to the second simpoint showed 32% of floating-point operations. However we did not run the second simpoint because the second simpoint requires a fastforward of 1.72 trillion instructions.

3.4 Physical Modeling

As CRIB requires significant modification to structures in the execution core, physical modeling is done to assess latency, area, and power consumption. We model each structure in the out-of-order window as well as structures in our CRIB execution core using a combination of synthesis and CACTI 5.3 [84]. Structures that require small stor-

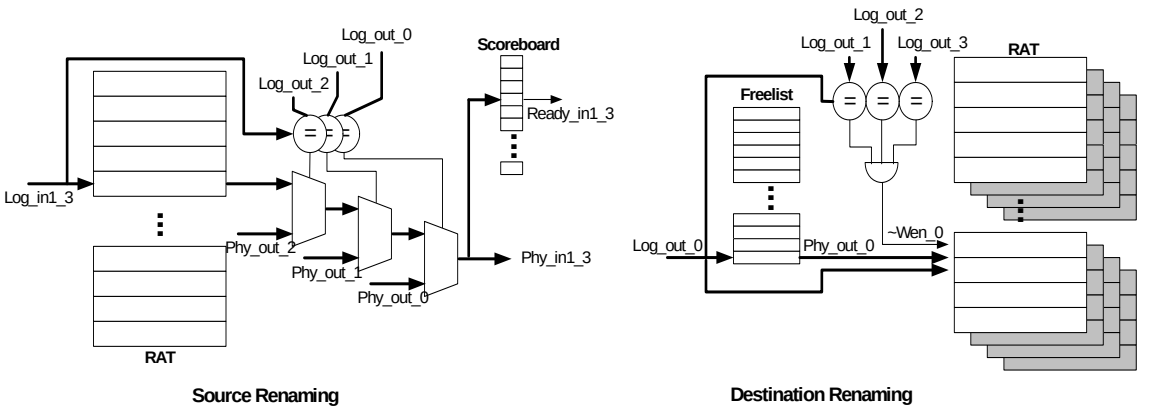


FIGURE 3-3. Register Alias Table. Instruction 0 is the oldest in the dispatch group while instruction 3 is the youngest. “

age with large combinational blocks, such as register alias table, reservation station, store queue, load queue, and ALU, are modeled using Verilog and synthesized using Synopsys Design Compiler with the TSMC tcbnptc 0.65 CMOS standard cell library. Large structures that consist mostly of SRAM such as caches, reorder buffer, immediate RAM, and physical register files are modeled using CACTI 5.3. The rest of this chapter describes how the structures are modeled.

3.4.1 Register Alias Table (RAT)

Register alias table (RAT) consists of a RAM-based map table with eight checkpoints, a scoreboard, and a physical register freelist. Figure 3-3 shows how the source renaming and the destination renaming are modeled. A source logical register is first used to access the RAT to get the physical source it is mapped to. The source logical register is also compared with all the older destination logical register in the dispatch group. For example, the second instruction has to do the comparison with only the first logical destination register, while the fourth instruction has to do the comparison with the first, second, and third logical destination registers. Staggered two-to-one muxes are used to choose

whether to use the mapping from the RAT or the new physical destination identifier from the earlier instruction in the group. The physical register identifier is then used to access the scoreboard to check the readiness of that particular physical register.

Destination register renaming is done by first accessing the physical register freelist to get a free physical register. It is then compared with all logical destination registers of the younger instructions in each dispatch group. For example, instruction 0 has to compare its logical destination register with instruction 1, instruction 2, and instruction 3. However, instruction 3 does not need to do any comparison since it is the youngest in the group. Any comparison match disables the write enable signal so that the physical destination register of the particular instruction is not written into the RAT. Instead, the youngest destination definition is the one stored in the RAT.

The RAT is a small RAM with multiple read ports and write ports. Each entry consists of five to seven bits of physical register identifier depending on the number of physical registers of the machine. The modeled RAT renames four instructions each cycle. Thus, it requires eight read ports for the source registers and four write ports for the destination registers. Several comparison circuits are added to correctly rename source and destination registers.

A freelist is a circular buffer with a head and tail pointer. It is written from the commit stage as physical registers are being freed. It is read from the rename stage to get a free physical register to do the renaming. There are as many entry in the freelist as the number of physical register minus the number of logical register. A scoreboard is used to maintain the readiness of each physical register. It is read at the rename stage and written at the write-back stage. Checkpoints are shown in Figure 3-3 as shaded area. A checkpoint

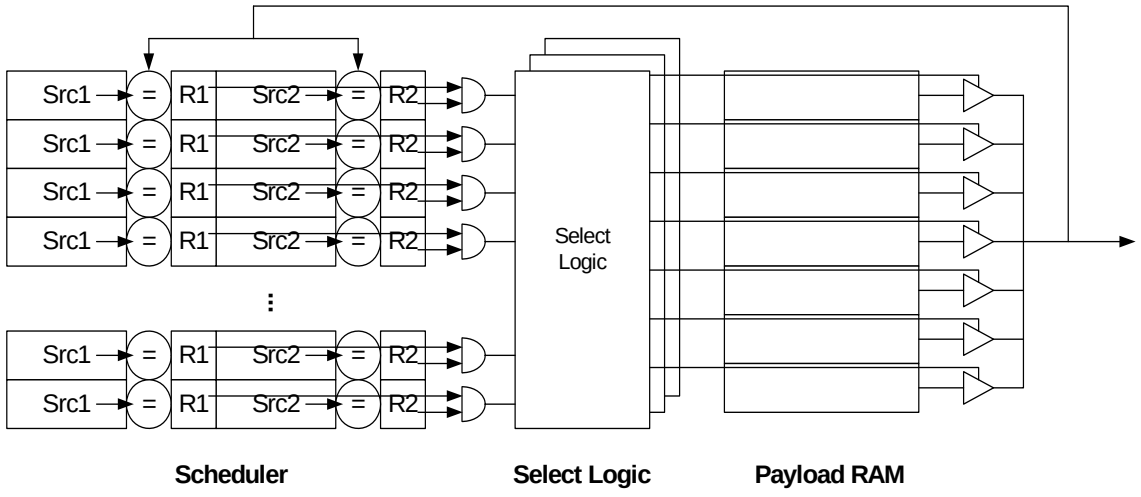


FIGURE 3-4. Reservation Station.

is created on low confidence branches. The RAT checkpoint is implemented as an eight-entry circular buffer with a head and a tail. Each entry of the checkpoint is wide enough to store the content of the current RAT. All structures in RAT are modeled using verilog. Two RATs are modeled, integer RAT and floating-point RAT.

3.4.2 Reservation Station

The reservation station buffers instructions that are not yet executed. Instructions are waiting in reservation station because of unready operands or issue bandwidth limitation. The reservation station includes scheduler, select logic, and payload RAM. The scheduler does tag comparison to determine if an instruction should be awakened upon a data broadcast. When both operands are ready, an instruction is marked as ready to be issued. Select logic chooses one of the ready instructions to issue. We modeled the select logic using a hierarchical priority encoder based on [67]. There are as many select logic structure as the issue width. Payload RAM contains information needed to do execution such as opcodes, destination tags, immediate values, reorder buffer indexes, and load/store queue indexes. The scheduler and select logic is modeled using verilog. Since payload

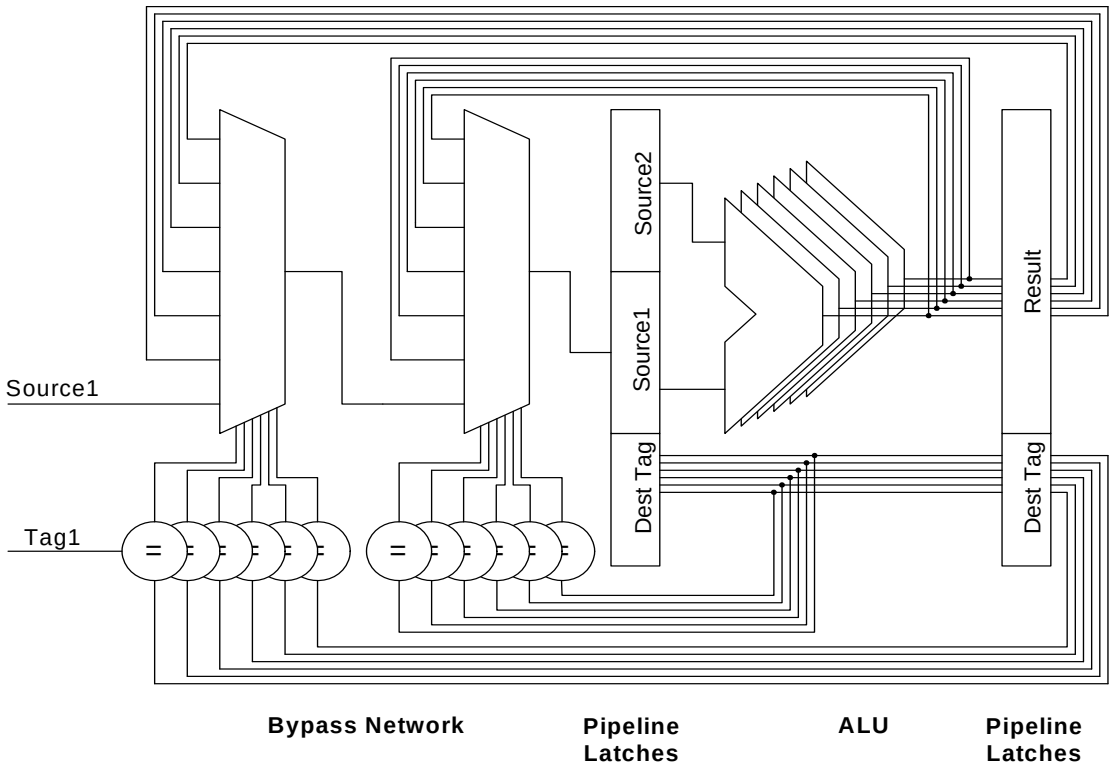


FIGURE 3-5. ALU and Bypass Network.

RAM is simply a direct-mapped RAM, it is modeled using CACTI.

3.4.3 ALU and Bypass Network

We modeled ALU and a two-level bypass network. A two-level bypass network is necessary because there is a register read stage in between issue and execution. Figure 3-5 shows the diagram of the ALU and two-level bypass network for one of the sources. In each level, the source tag is compared with destination tags from all execution lanes to determine if the source operand should be picked from the bypass network instead of from the register file. The first level of the bypass network tries to catch the output from instruction currently writing their results back into the physical register file. The second level tries to catch the output from instruction currently in the execution stage. Each execution lane has four sets of six comparators (corresponding to six issue width) and two seven-to-

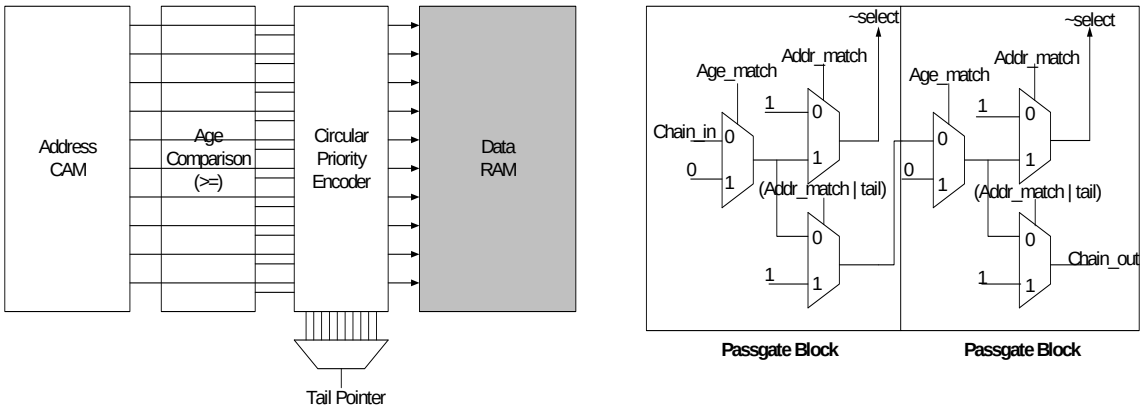


FIGURE 3-6. Load / Store Queue. The shaded data RAM is only for store queue

one muxes, corresponding to two operand sources.

3.4.4 Load and Store Queue

Figure 3-6 shows the modeling of load and store queues. Load and store queues consist mainly of address CAM, an age comparator, and a circular priority encoder. A store queue would also have additional data RAM to store the store-data. When a load instruction needs to check for any matching stores in the store queue, it accesses address CAM and age CAM in parallel. Address comparison signals and age comparison signals are sent to the priority encoder to set the matched signals for only older matching stores/loads. The priority encoder then chooses the youngest among the older matching stores/loads. The circular priority logic is modeled based on circuitry patented by Intel [19]. It consists of a circular chain of passgates as shown in Figure 3-6. Chain_out signal from one passgate block is connected to the chain_in signal to the next passgate block in circular manner. The chain of passgates starts to evaluate starting from the age matched entry that signifies the beginning of older stores that the load has to check. It stops upon reaching the first address matched entry, the youngest matching stores among all the older

stores. Address CAM and data RAM are modeled using CACTI while the priority encoder and age comparison logic are modeled using verilog.

3.4.5 Reorder Buffer

The reorder buffer (ROB) is modeled as a 4-wide circular buffer to minimize the number of ports; only one write port is needed to insert four instructions into the window and one read port is needed to commit four instructions from the window. Each ROB-entry contains four instructions. Each instruction entry has 100 bits, consists of:

- Valid bit - 1 bit
- Instruction PC - 64 bits
- Opcode - 7 bits
- Output logical register - 5 bits
- Previous output physical register - 6/7 bits
- Load/store queue index - 6/7 bits
- Checkpoint index - 3 bits
- Various status bits - 6 bits

Different bits of the reorder buffer are accessed in different pipeline stages, i.e. status bits are inserted on writeback stage while other bits are inserted on dispatch stage. Thus, the ROB is banked according to the access stage. The reorder buffer is modeled using CACTI.

3.4.6 Register File

Integer physical register file is modeled with four read ports and four write ports since we assume that port reduction techniques [68] are commonly used in today's micro-

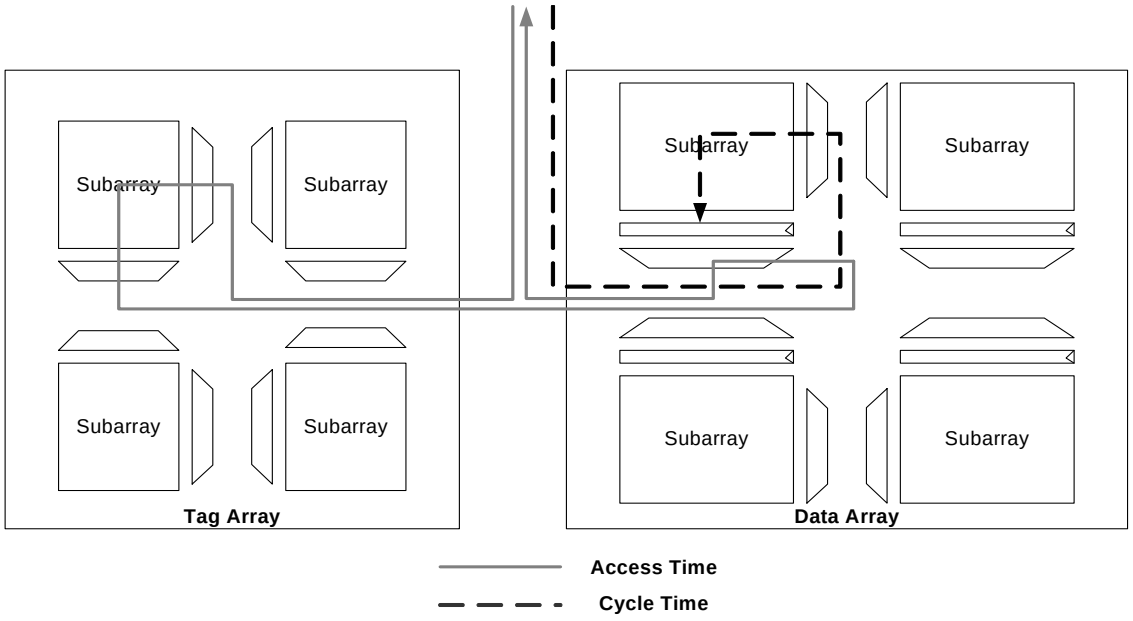


FIGURE 3-7. Data Cache.

processors. Although port reduction is assumed for the physical model, our architectural models for the baseline machine assume an ideal physical register file. Thus, there is no stalling when both operands need to be read from the physical register file. Floating-point physical register file physical model also assumes port reduction technique with two read ports and two write ports. Both the integer and floating-point physical register files are modeled using CACTI.

3.4.7 Data Cache

Data cache is modeled using CACTI as a one port pipelined cache with no banking. Figure 3-7 shows the access time path and the cycle time path that is used in the model. The total access time of the cache is shown in the solid grey line while the path used to determine the cycle time is shown in dashed black line.

$$AccessTime = max((tagArrayAccess + dataArrayRouting), dataArrayBeforeMux) + muxDelay + dataArrayOutputDriver + dataArrayRouting$$

Cycle time is determined using the longest path where no latches can be inserted. In this case the data array path starting from routing all the way to the sense amps are chosen. The tag array result is latched inside next to the mux inside the data array. In the corresponding cycle, the data is muxed and routed out.

3.5 Performance Debugging

CRIB implementation in our timing simulator requires major changes due to the removal of renaming, as well as the full reconstruction of wakeup-select-issue-execute stages. It is possible to insert sanity checks, in form of assertion statements, in various places in the simulator, to mostly eliminate correctness bug. However, a different methodology to do performance debugging is necessary.

For performance debugging purpose, an IPC tracing option is added so that the simulator reports IPC every million of instruction when the option is enabled. The IPC report of CRIB can then be compared with the IPC report of the baseline machine to find out which of the million of instruction group shows suspicious speed-up or slow-down. A detail tracing option can then be enabled to print out instructions traces for the particular instruction group for both the baseline machine and CRIB. The information for each instruction includes tag, PC, source and destination logical registers, dispatch cycle, issue cycle, and commit cycle.

A post processing script reads the trace reports and compares the commit cycle between the two reports.The script also creates buckets for various slippage in commit cycle, i.e. 1-5 cycles, 5-20 cycles, 20-50 cycles, > 50 cycles. A slippage in commit cycle occurs whenever the two instructions from the two reports take different number of cycles

to commit the next instruction. If a slippage occurs, the PC of the instruction is thrown into the appropriate slippage bucket. If the same PC is already in the bucket, the counter for the PC is incremented. At the end of the processing, the PCs in each of the bucket is dumped together with their counter values.

From the output of the post processing script, the problematic PCs can be noted. A manual debugging can then be done focusing on the problematic PC. Source code obtained using *objdump*, is also proved useful in aiding the manual debugging of the two traces. Many of the benchmarks execute a single loop for millions of dynamic instructions. Thus, whenever the problematic PC is found, one can easily do manual execution of these instructions and compare it with the instruction trace to identify the problem.

3.6 Chapter Summary

We use an x86 simulator derived from Bochs to collect the data presented in this thesis. Two baseline machines are used to compare against. Baseline 1 is a small out-of-order core with 24-entry ROB while baseline 2 is large out-of-order core configured to represent today's microprocessor configuration. SPEC CPU2006 Integer and Floating Point are used as our primary metric to evaluate CRIB. Redundancy analysis is used to choose eight benchmarks each from SPEC INT 2006 and SPEC FP 2006. Simpoint is used to pick a representative portion of the benchmark. Benchmarks are fast-forwarded to the first simpoint before timing analysis is performed.

Physical modeling of different structures in baseline machines is done to assess latency, area, and power consumption. Structures with small storage and large combinational blocks are modeleled using Verilog and synthesized using Synopsys Design Com-

piler. Large structures that consists mostly of SRAM are modeled using CACTI.

Lastly, performance debugging methodology is developed to catch performance bugs. Sanity checks in form of assertion statements eliminates correctness bug. However, since CRIB implementation requries major changes in the timing simulator, a different methology for performance debugging is necessary.

Chapter 4

CRIB: Consolidated Rename, Issue, Bypass

This chapter proposes consolidation of rename, issue, and bypass into one structure, CRIB, for a more power efficient out-of-order processor. It first describes the basic concept of CRIB in Section 4.1. Section 4.2 elaborates on how the CRIB concept is expanded into a full out-of-order execution core. Section 4.3 explains design extension for the CRIB by employing deep-pipeline. Section 4.4 elaborates data cache optimization that can be employed together with CRIB.

4.1 CRIB Concept

As shown in Chapter 1, explicit register renaming requires multiple structures, such as the register alias table (RAT), the reorder buffer (ROB), the register file (RF), and the reservation station (RS), to support operand delivery. Tags and data are moved from one structure to another while status bits are updated in each of the structures, resulting in significant power consumption.

In CRIB, we propose to remove redundant structures and activities related to register renaming algorithm by consolidating rename, issue, and bypass into the CRIB. Inside the CRIB, conventional register renaming is replaced with positional renaming to detect true data dependencies and to relax false data dependencies, enabling out-of-order execution. Decoded instructions are put into the CRIB where they read their source operands and compute results. Upon completion of the CRIB, the results are latched back into the register file. The energy consumption of CRIB is mostly due to execution energy as

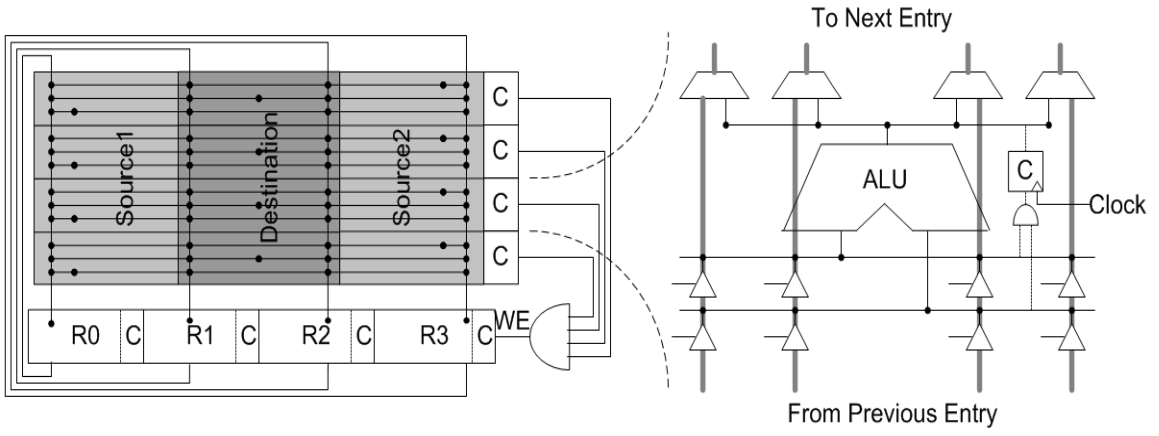


FIGURE 4-1. CRIB Concept. The left figure shows four-entry CRIB. Each register has its own path going across the CRIB, connected to ALU in each entry. The right figure shows one CRIB entry

opposed to operand delivery overhead as in conventional out-of-order design.

The concept of the CRIB is shown in Figure 4-1. The left figure shows the CRIB while the right figure shows one CRIB entry. For simplicity, the figure assumes four logical registers and a four-entry CRIB. Each logical register has its own column that spans the CRIB. The CRIB is a connection matrix connecting logical register columns, which are spaced horizontally, to an ALU. It also connects the ALU result back to the appropriate register column. Each instruction in the CRIB taps its source operands from the register columns. It then overwrites its destination register column accordingly. As each CRIB entry has an ALU, it is not necessary to latch the execution result. Instead, the register value propagates through the columns. As shown in Figure 4-1, tapping source operands and overwriting destination registers can be implemented using tri-state buffers and two-to-one muxes.

In addition to the core resources, there is a simple architected register file or ARF, which is really just a rank of latches. Once inserted, instructions stay in the CRIB until all entries finish executing. When all completion bits in all entries are set, the ARF is clocked

and results are latched.

Data propagation inside the CRIB is done combinationally without latching. It is made possible since an ALU serves the instruction until it leaves the CRIB, when the result is then latched in the ARF. Although data propagation inside the CRIB is done combinationally, instruction wakeup and completion are fully synchronous. It is done by adding a completion bit to each entry of the CRIB and to each register column. When registers have their completion bit set, it means that they are ready to be consumed. As instructions are dispatched into the CRIB, they unset the completion bit of their destination registers, hence preventing their consumer from consuming the register value. When both source operands are ready, each instruction set its own completion bit that will propagate to its destination register. Figure 4-1 assumes one-cycle execute instructions, which set their completion bit in the very next cycle after both operands are ready. When an instruction needs multiple cycles to complete, the completion bit is set accordingly.

Figure 4-2 shows an example of how the CRIB resolves read after write (RAW), write after read (WAR), and write after write (WAW) dependencies. A list of instructions placed in the CRIB is shown with the chronological order going up. Starting at the top left, and proceeding clockwise, each instruction is inserted into the CRIB. The connections that are formed are shown in the same color and texture as the instructions. The top left diagram shows (in solid blue line) source registers R0 and R3 being routed from their logical columns to the ALU of the first entry, with the result routed to the logical column for R2. The top right diagram shows how RAW and WAW dependencies between the first and the second instructions are resolved. The second instruction simply taps the R2 value produced by the first instruction as one of its sources. It then overwrites the R2 column with

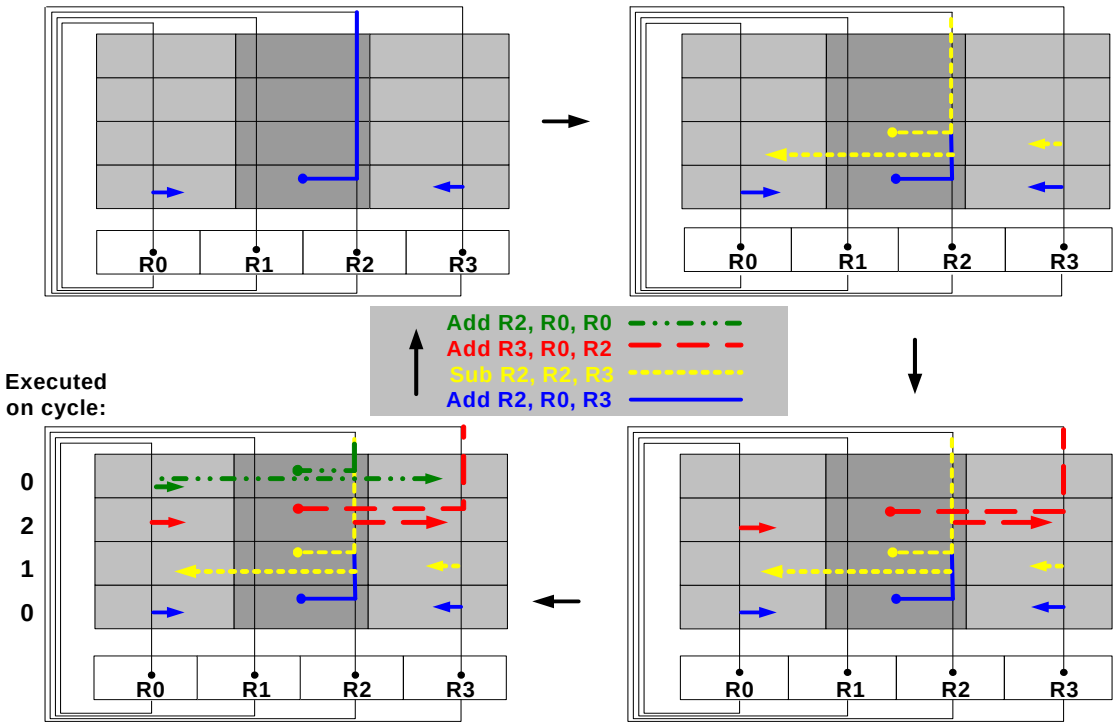


FIGURE 4-2. CRIB Example. Time progress clockwise from top left, with each figure adding the next instruction in the code snippet to the CRIB. The last diagram (bottom - left) also shows when each instruction gets executed

its own result. The bottom right diagram shows how the WAR dependency is solved between the second and the third instructions. Due to the positional ordering of instructions in the CRIB, the third instruction can overwrite R3 without worrying if the second instruction is done reading it. The bottom left diagram shows the final connections among these four instructions.

Although shown step by step, all the connections form in parallel when instructions are inserted into the CRIB. Instructions can then start issuing and executing, subject to the readiness of their operands. Figure 4-2 illustrates that although the fourth instruction is the youngest one, it starts executing in the same cycle as the first instruction as their operands are ready on insertion.

The execution core in the proposed CRIB consists of only a CRIB and an archi-

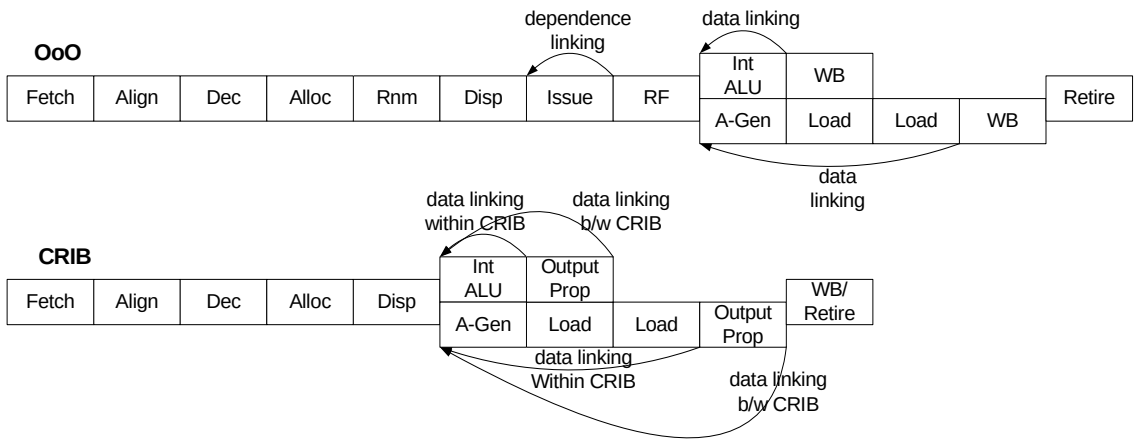


FIGURE 4-3. CRIB Pipeline Diagram.

tected register file (ARF) as shown in Figure 4-1. Each CRIB entry is a connection matrix connecting logical register columns to an ALU and connects the result back to the appropriate register column. The architected register file is a rank of flip-flops with write-enable capability organized into register columns. The CRIB, together with the register columns provides positional renaming to solve data dependency, wake-up and issue logic, as well as bypassing capability for instructions in the CRIB. In CRIB, execution occurs out-of-order subject only to true dependencies.

Figure 4-3 compares CRIB and baseline out-of-order pipeline diagrams. In the front-end, CRIB removes one stage, the rename stage. In the execution window, CRIB consolidates the issue, register file read, and execute stages into one. Lastly, CRIB also combines writeback and retire stages. When a dependent instruction is in the same CRIB partition (partitions are explained in Section 4.2) as the producer, they can issue back-to-back. On the other hand, output propagation cycle(s) have to be added when dependences cross CRIB partitions. When all instructions finish executing, the results are latched into the ARF and the instructions are retired from the CRIB.

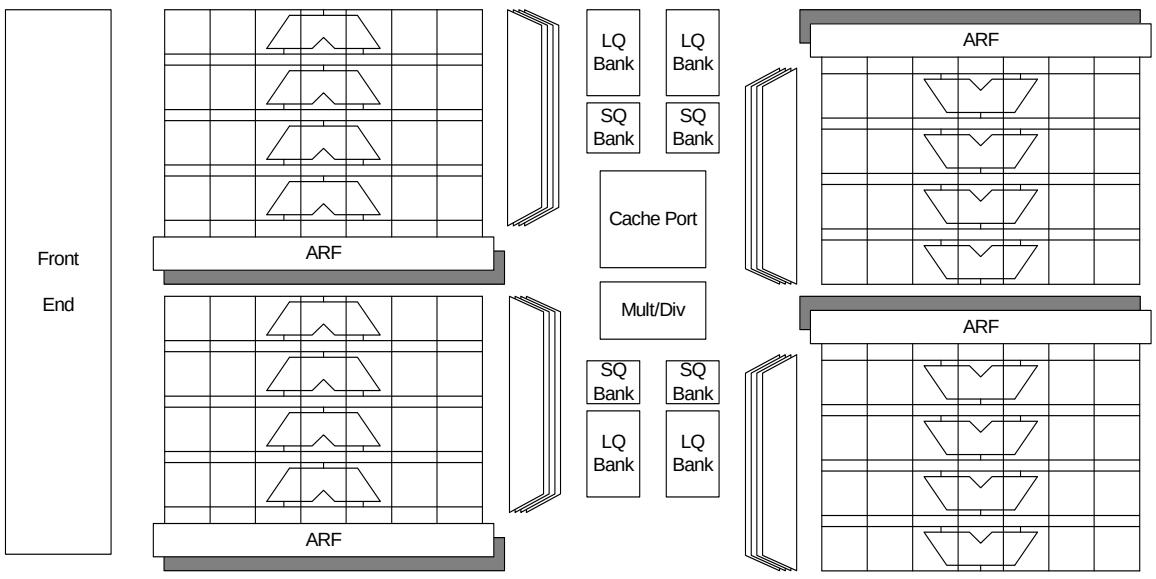


FIGURE 4-4. Four-Partition CRIB.

4.2 CRIB as a Realistic Execution Core

Section 4.1 assumes a simplistic view of CRIB with only simple integer instructions. In this section, the CRIB concept is expanded into a realistic execution core. With only the four entries described in Section 4.1, CRIB will have a very limited reach for extracting instruction level parallelism. Thus, scaling up CRIB is necessary for high performance. Section 4.2.1 illustrates how CRIB is scaled up into partitioned CRIB. Load and store instruction handling is described in Section 4.2.2. Section 4.2.3 and Section 4.2.4 explain branch misprediction handling and the floating point unit.

4.2.1 Partitioned-CRIB

Scaling up CRIB is crucial to extract instruction level parallelism and achieve high performance. One way to scale up the CRIB is by simply increasing the number of CRIB entries. However, because all instructions have to stay in the CRIB until all of them finish executing, simply increasing the number of CRIB entries will lead to low utilization and

will delay dispatch of later instructions into the window.

Another way to scale-up the CRIB is by replicating the CRIB into a partitioned-CRIB as shown in Figure 4-4. The partitioned-CRIB is maintained in a circular fashion. Instances of the architected register file are inserted between CRIB partitions. Only the architected register file at the head of the partition has the committed state of the program. As a partition completes execution, the head or commit pointer is moved to the next architected register file instance. To allow register values to travel through the architected register file to the next partition without getting latched, transparent flip-flops are used [38][43]. The architected register file flip-flops that are not holding the committed state of the program are left transparent, reducing power consumed for clocking latches. Ready instructions in CRIB partitions can evaluate concurrently, exposing additional parallelism. Data values take an additional cycle to travel from one partition to the next, adding a cycle of delay between dependent instructions that are in adjacent CRIB partition. Although data is not latched, complete signals are always latched every cycle to ensure a fully synchronous design. A partitioned-CRIB with a few entries per partition can prevent dispatch bottlenecks due to improved utilization, but it incurs extra cost due to additional architected register file latches between partitions. Sensitivity to the size and the number of CRIB partitions is explored in Chapter 6.

Figure 4-5 shows how the complete signal travels from one CRIB partition to the next. Architected register file consists of data portion, which use transparent flip-flops, and completion bits, which use regular flip-flops. Each CRIB entry also has its own completion bits. In the example, instructions in CRIB 0 are inserted at cycle 0 while instructions in CRIB 1 are inserted at cycle 1. Before instructions in CRIB 0 are dispatched, R2

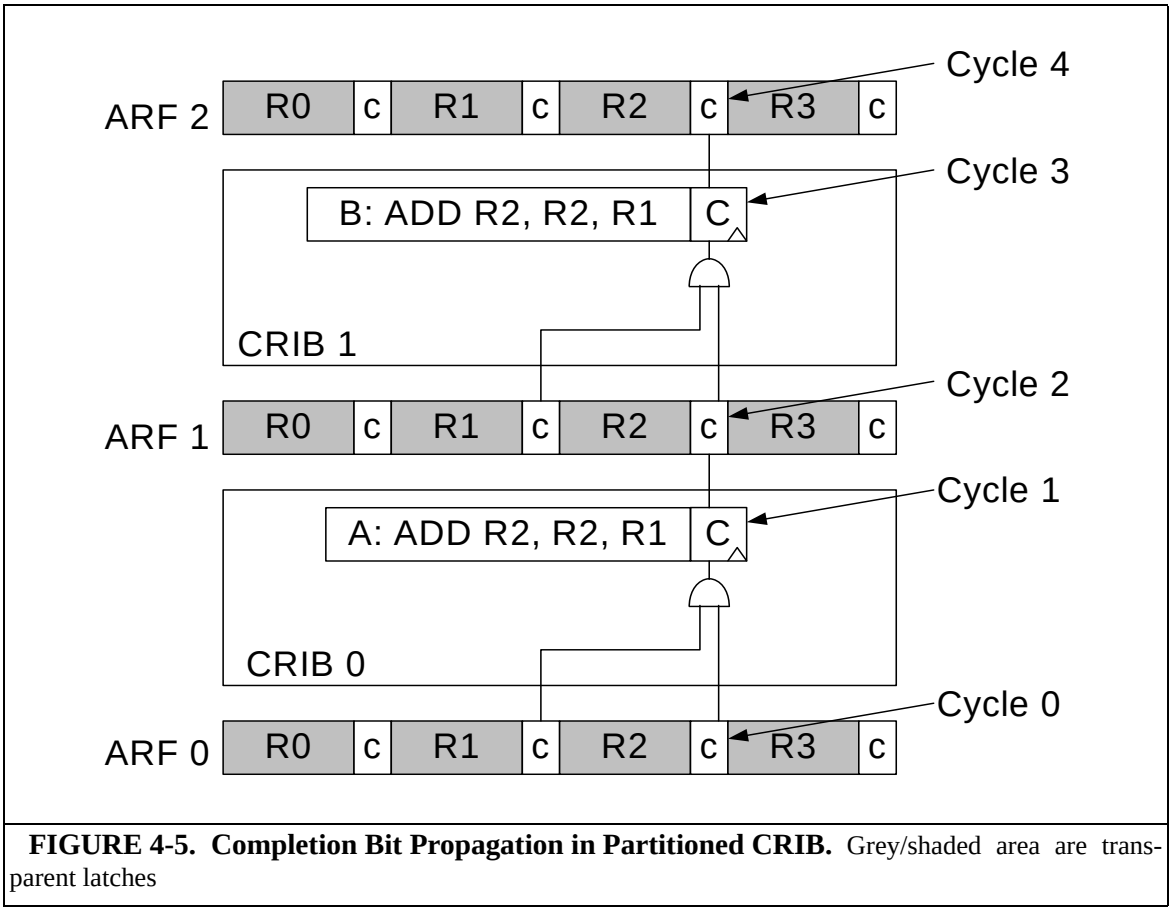


FIGURE 4-5. Completion Bit Propagation in Partitioned CRIB. Grey/shaded area are transparent latches

in both architected register files are marked as complete. As the instructions are dispatched at cycle 0, the following happens.

Cycle 0:A is dispatched to CRIB 0 and clears its completion bit.

ARF 0 serves as the head of the CRIB

Cycle 1:As R2 is ready, A executes and sets its completion bit.

The completion bit of R2 at ARF 1 is unset

B is dispatched to CRIB 1 and clears its completion bit.

Cycle 2: A's completion bit propagates,

R2's completion bit at ARF 1 is set.

The completion bit of R2 at ARF 2 is unset.

Cycle 3: B executes and sets its completion bit.

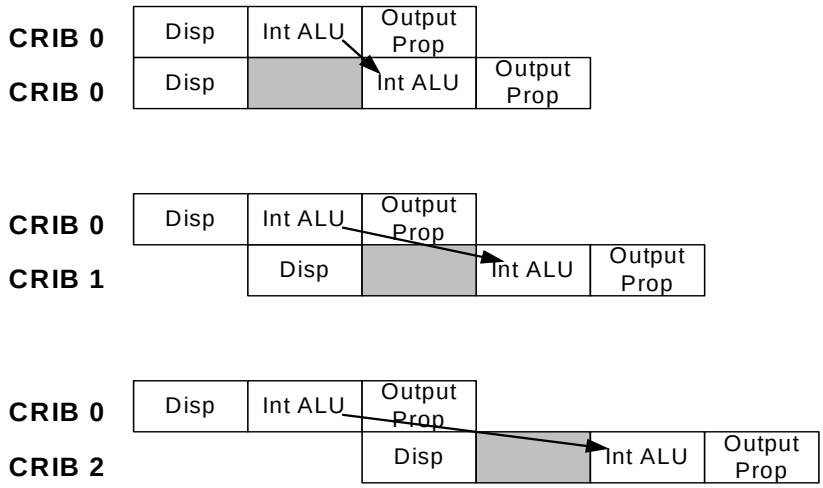


FIGURE 4-6. Producer-Consumer Pipeline Diagram.

Cycle 4: B's completion bit propagates and

R2's completion bit at ARF 2 is set.

The above examples assume more than two CRIB partitions. In the case where there are only two CRIB partitions, what happens at cycle 4 changes slightly since there is no ARF 2. Two cases can happen. In the first case, all instructions in CRIB 0 execute at cycle 1. Hence they are all written back and retire at cycle 2 and the head is now ARF 1. Thus in cycle 4, B's completion bit propagates back to ARF 0. The other case happens when the head of the CRIB is still ARF 0. In this case, B's completion bit will not propagate back to ARF 0, until the head of the CRIB moves to ARF 1.

Figure 4-6 summarizes the producer-consumer pipeline in CRIB. When both producer and consumer happens to be in the same CRIB, execution happens in back-to-back cycles. If the consumer happens to be in the next CRIB, the data takes one cycle to propagate to the next ARF and the consumer can execute the next cycle. If the consumer is located further way, i.e. the producer is at CRIB 0 while the consumer is at CRIB 2, the

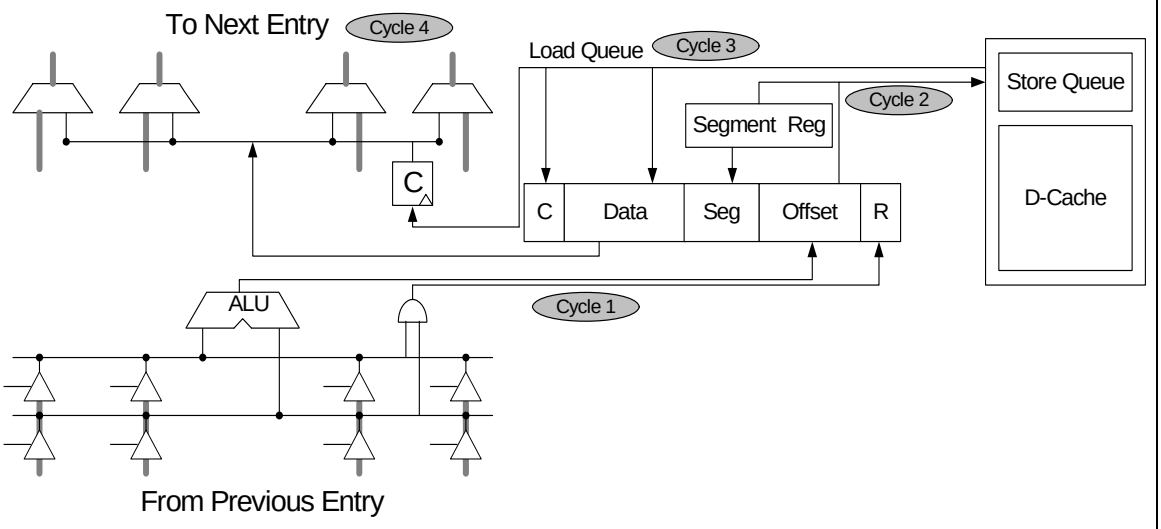


FIGURE 4-7. CRIB Load Instruction Handling. Assuming 2-cycle d-cache latency

data takes two cycles to propagate: one cycle to propagate to ARF 1 and another cycle to propagate to ARF 2. Then, the consumer can execute in the following cycle.

So far simple integer instructions are assumed, so resources are easily replicated in each CRIB entry. However for other instructions, replication is too expensive and a different approach is needed. The next few sections will explain how different instructions are handled in CRIB. For simplicity of explanation, a four-entry partitioned CRIB is assumed.

4.2.2 Memory Instructions

As explained earlier, the CRIB entries do not latch instruction results; instead each CRIB entry's ALU keeps driving its result into its logical register destination column. However, applying the same approach for load instructions would require holding a cache port for the entire evaluation of a CRIB partition, and would create too much demand for cache ports. Instead, CRIB uses the load queue to latch the data and holds the read port to the load queue as long as load instructions stay inside the CRIB.

Figure 4-7 shows how a load instruction is handled in CRIB. As a load instruction

is dispatched into the CRIB, it is assigned a load queue entry. When the load instruction becomes ready, address generation is done inside the CRIB entry using the integer ALU. At the end of the cycle, the address is latched into the load queue and the ready bit is set. A simple FIFO arbiter checks the load queue every cycle to find ready instructions to be executed. At the next cycle, the arbiter picks the load instruction. The load is then executed as in a conventional machine where it access the cache and associatively searches the store queue. For architectures with a segment register like x86 architecture, the segment register is also read and combined with the offset to create the memory address. The segment selector is also written to the load queue to create the full memory address in the load queue. Once the load fetches the data from either the store queue or cache, the data is then written to the data portion of the load queue. A completion bit in the load queue entry as well as the one in the CRIB are then set. To drive the data up the destination register wire, the read port for the load queue data is held open as long as the load resides in the CRIB.

To avoid an excessive number of full read ports, the load queue is banked to match the number of CRIB partitions. Each load queue bank only needs as many read ports as the number of load instructions in each CRIB partition. Because load instructions compose 40% - 50% of total instructions on average in our workloads, we limit the number of load instructions in each CRIB partition to two. This kind of limitation (the number of memory instructions per dispatch group), is not uncommon and is widely adopted in current microprocessors [83][47]. When the limit is reached, the dispatch logic stalls. A similar limitation is also imposed on store instructions; only one store is allowed per CRIB. When a store instruction is ready and issued, it writes its address and data into the store queue. When the CRIB partition finishes, the store address and data are sent to the write buffer or

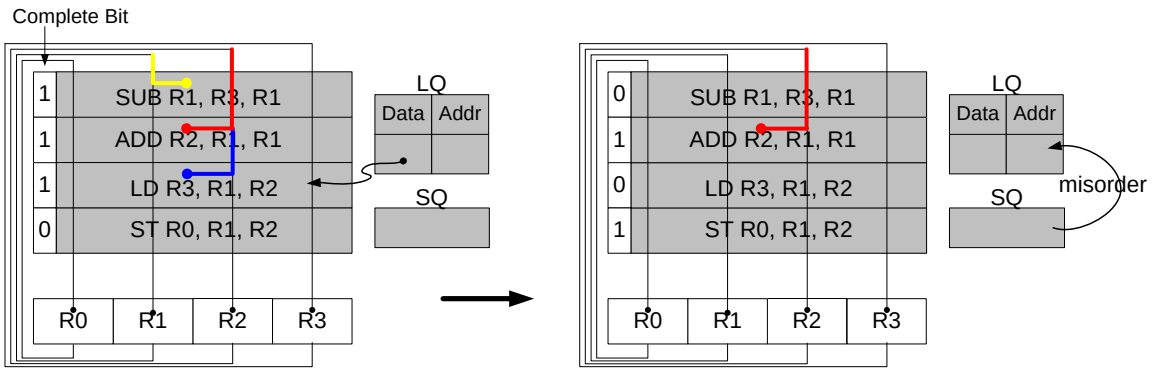


FIGURE 4-8. Recovery Example on Load-Store Misorder.

cache.

In CRIB, load and store instructions are ordered aggressively as in out-of-order machines. Ready younger load instructions executes even when there is an older incomplete store instruction. While a flush recovery is required in conventional out-of-order machines, a simpler re-execution is enough for CRIB. As shown in Figure 4-8, an invalidation signal from a mis-ordered older store results in the violating load queue entry deasserting its completion bit. The deassertion of the load instruction completion bit propagates to to its dependent, the SUB instruction in the Figure 4-8, and triggers the deassertion of the dependent's completion bit. Once the load instruction retrieves the correct data, the completion bit is asserted once again. As illustrated in Figure 4-8, the ADD instruction does not need to deassert its completion bit because it does not depend on the output of the load instruction. In order to avoid any invalid latching, a store instruction has to invalidate all misordered load instructions before it sets its completion bit.

This recovery mechanism is very lightweight, so CRIB does not need to use any memory disambiguation predictor, such as Alpha style [47], or store set [20], for its load-store reordering. Instead, CRIB can simply assume that no misordering will happen and

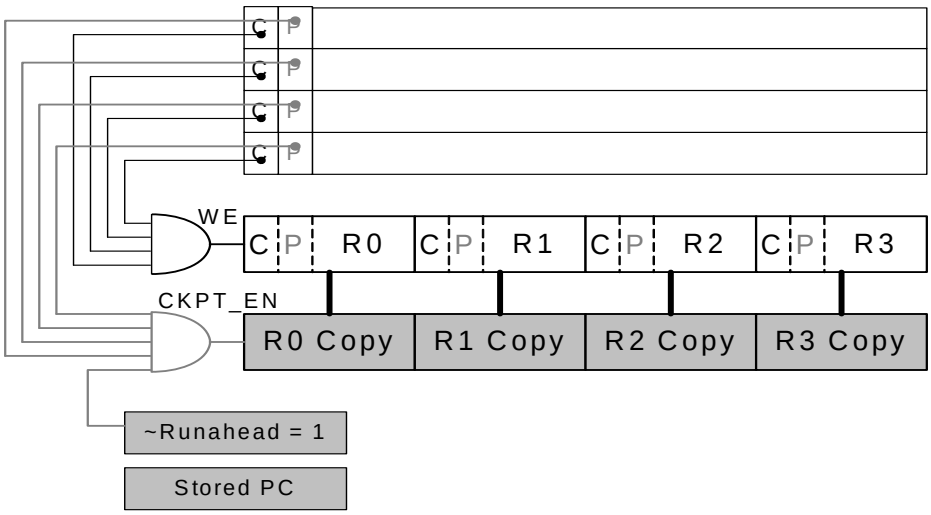


FIGURE 4-9. CRIB with additional checkpointing capability. Added checkpointing logic is shown in grey color

issues load instructions as soon as they are ready. This aggressive reordering can be especially advantageous when typical predictor produces many false hits and reduces the potential parallelism that can be extracted from this non-aliasing load instructions.

Instructions are allowed to retire from a CRIB when all of them have finished executing. This means that a level-2 cache miss can stall the CRIB for a long time, and it is very costly. In order to avoid stalling the CRIB on level-2 cache misses, a simple runahead mechanism [22][64] is used. To support runahead, a second copy of the ARF is needed and a poison bit is added to each of the register column and to each of the CRIB entries. When a load instruction detects that it has a level-2 cache miss, it assumes completion by setting its completion bit. In addition to that, it also sets its poison bit since its data is not valid. The poison bit propagates in a similar manner as the completion bit. When all instructions complete, the ARF is clocked . Checkpointing logic detects that this is the first time some poison bits are detected and triggers an ARF copy to its shadow copy. Thus the previous ARF value is now copied in the shadow copy and the ARF now contains the

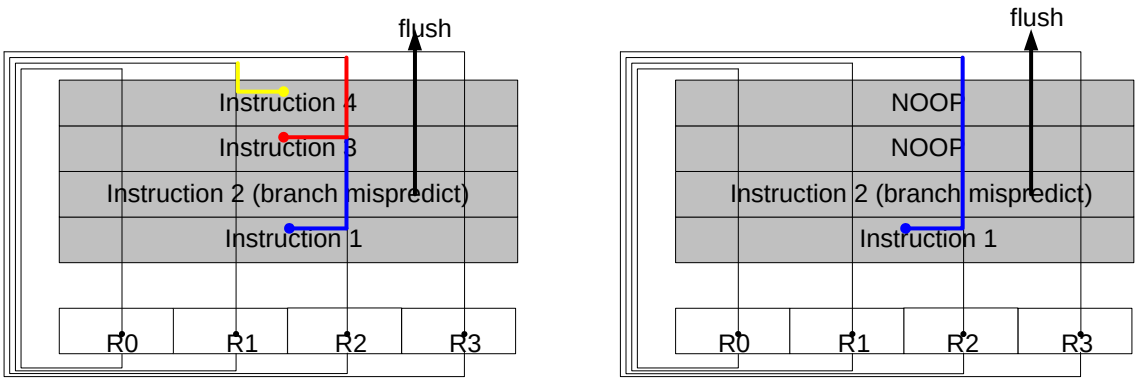


FIGURE 4-10. CRIB branch misprediction recovery.

new and poisoned value. The PC address of the first entry of the CRIB is also recorded in the control unit. The pipeline proceeds as before. When the missed load returns, the CRIB is flushed. Fetch is redirected to the stored PC, and the checkpointed ARF is restored.

In CRIB runahead, poisoned runahead load instructions are not sent to the cache as the address might be invalid. For simplicity, store instructions under runahead mode are thrown away, since we did not observe significant benefit from a runahead cache [64].

4.2.3 Branch Instructions

Without explicit register renaming, branch misprediction handling in CRIB is much simpler than in conventional out-of-order machines. Branch misprediction handling in conventional out-of-order machine varies from register alias table (RAT) checkpointing, reorder buffer unrolling, and retirement RAT mechanism. Retirement RAT is the simplest and slowest among the three. RAT checkpointing provides a faster but more complex handling. Checkpoint creation for all branch instructions in the window is quite costly. Thus, a limited number of checkpoints are usually used. To avoid excessive correct-path re-execution, checkpointing can be used together with reorder buffer unrolling.

In CRIB, branch misprediction and precise exception handling is much simpler

and precise. On branch mispredictions, the mispredicted branch drives a global signal up the CRIB that forces youngest instructions to transform into NOPs. The NOPs stop overwriting their logical destination registers. Once execution finishes, the ARF captures the correct state of the program at the branch boundary, as shown in Figure 4-10. A few additional cycles are needed to cleanup the store queue and the load queue from younger instructions. The same approach can be used for precise exception handling.

4.2.4 Complex Integer Instructions

Complex integer resources, such as multiplication and division units, are shared since they are too expensive to replicate. The units are pipelined as in a conventional machine. However, since CRIB does not latch execution results, the final pipeline stage has to be occupied by the instruction until the result is latched back into the ARF. Once the result is written back, the next instruction in the pipeline can then move to the last stage and drive its output to its destination register column. To avoid deadlock, complex integer instructions per CRIB are limited to the number of units available and complex instructions have to be issued in order. In other words, if there is only one multiplication unit, the number of multiply instruction per CRIB is limited to one.

4.2.5 Floating Point Instructions

Floating point instructions are handled in the same way as integer instructions. A floating point CRIB is placed side by side with the integer CRIB. This floating point CRIB is able to handle floating point execution and other non-integer execution such as SIMD extensions. If a partition contains floating point instructions, they are placed into the floating point CRIB rather than in the integer CRIB. Simple integer SIMD units are replicated

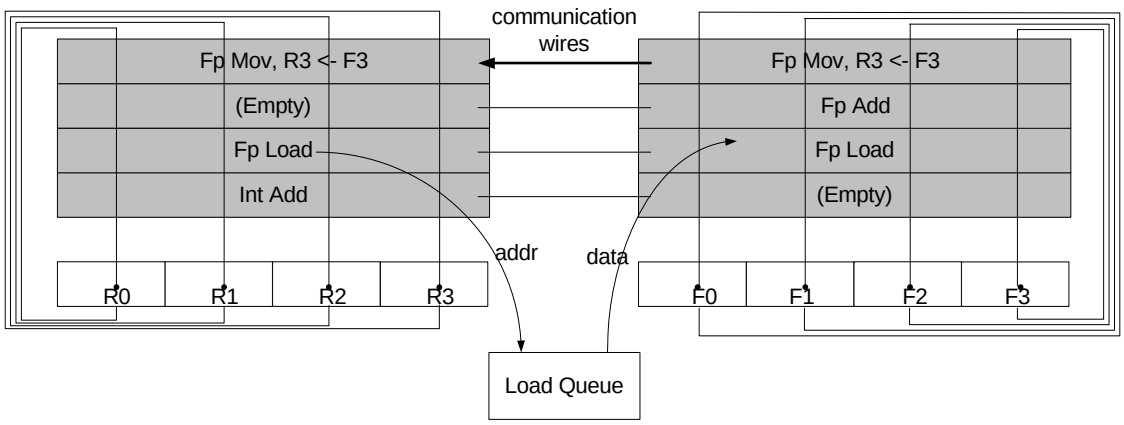


FIGURE 4-11. Floating Point CRIB.

for each entry of the floating point CRIB. FP units such as addition, multiplication, division, square-root, and others are shared in the same way as the complex integer units in the integer CRIB. A similar limitation to the number of FP instructions in a CRIB partition is also imposed, i.e. if there are two FP add-multiply, one FP division, and one SSE multiplication/division, then the number of FP add-multiply instructions in the CRIB is limited to two, and FP division instruction is one, and so on. Communication between integer and FP registers occurs through load and move operations. Communicating instructions reside in both integer and floating point CRIBs. The communication itself is done via the load queue or additional links, as shown in Figure 4-11. If the communicating instruction is a load, the address calculation is done in the integer CRIB, but the load result is read by its floating point counter part. For a move instruction, one half of the instructions does the input routing while the other half does the output routing. The data itself is sent using the communication links. There is a pair of communication links for each entry, one for each direction.

4.2.6 Miscellanous Details

While most instruction handling has been explained, there are some miscellaneous issues specific to x86 architecture. In this section we address these miscellanous details.

Temporary registers and atomic commit. X86 instruction set requires micro-op translation. In addition to the architected registers, there are temporary registers needed for communication between micro-ops. In CRIB, temporary registers are treated as one of the architected registers. A register column is allocated for each of the temporary register. Micro-ops can also be safely split across partitions. This is not a problem, as long as the micro-ops overwriting architected registers (as opposed to temporary registers) reside in the youngest partitions. In other word, the micro-op writing the architected register should be the last micro-op of that particular instruction. Thus, if an exception occurs and the last micro-op that writes to the architected register has to be flushed, only the temporary registers have been affected in the ARF, leaving architected state that is precise at the preceding macro-instruction boundary.

Partial register writes. In a conventional out-of-order x86 machine, partial register writes is an issue that requires a complex solution. Due to the fact that the write only happens to part of the register, the portion of the register that is not being overwritten has to be retrieved from the older definition of the register. Thus, the partial write instruction has to first read the older value of its definition register, reads its own sources, performs the necessary computation, concatenates the older value with the newer value, then writes it back. Although these steps can be implemented in several different ways, it is rather complex and expensive. Unfortunately, this partial register writes are often used for SSE instructions. SSE register width is 128-bits. However, load instruction width is only 64-bits. It means that two load instructions are needed to fill one SSE register, each of them

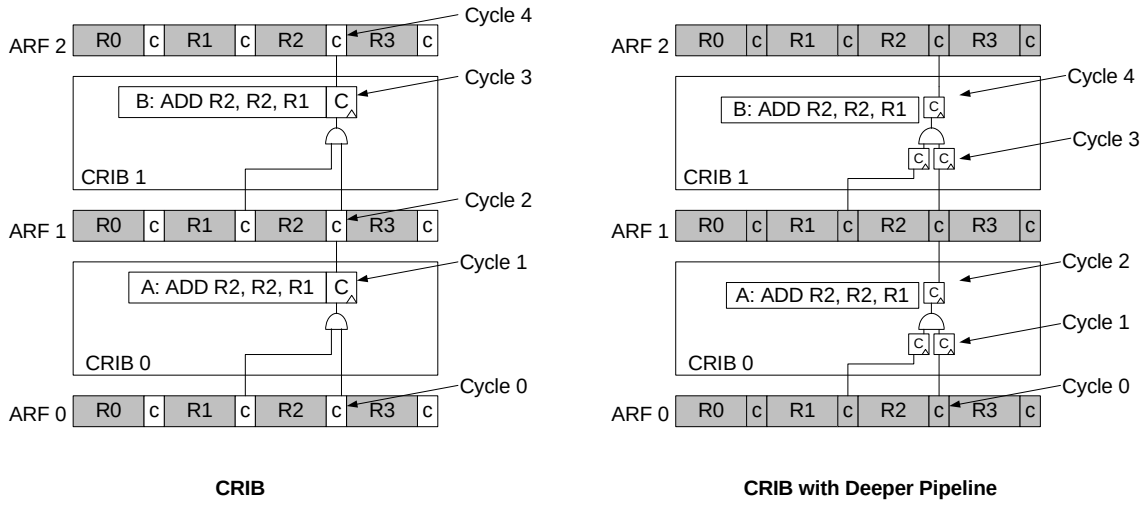


FIGURE 4-12. CRIB with Deeper Pipeline.

write a partial result into the register.

On the other hand, partial register writes are trivial to implement in CRIB. The CRIB entry that needs to do the partial write can simply choose to overwrite the part that needs to be overwritten and propagate the rest of the value from the older value. CRIB can also be expanded to treat each portion of the register as a separate register column and assign a completion bit to each portion. While it requires additional cost, it can be beneficial on certain programs where partial register reads and writes happen frequently, such as some floating point benchmarks presented in Chapter 6.

4.3 CRIB with Deeper Pipeline

In a realistic design, area constraints will limit the number of CRIB partitions to a small enough number that CRIB occupancy will stall dispatch. Since A CRIB partition is not reclaimed until all instructions in it have finished, the occupancy of that partition is determined by the latency of the longest path through the partition's dependence graph.

One way to reduce that latency, and improve occupancy, is to deeply-pipeline the CRIB hardware and clock it at a faster rate. Deeper pipelining comes at a high cost in a conventional out-of-order machines, since the data has to be latched in every pipeline stage, leading to additional area and power. But more importantly, the latch latency overhead, latch setup time, and clock propagation, reduces the benefit of increased clock rate. As the pipeline grows deeper, the total latency to finish the same computation increases due to latch overhead.

Since CRIB only latches the control/status bits, which are few and off the critical path, CRIB is able to implement deeper pipelining without suffering the same overhead as conventional machines. Figure 4-12 shows completion bit propagation for CRIB and the modifications necessary to implement a deeper pipeline. The shaded areas are transparent latches. As seen in the left picture, the data propagation path in CRIB mostly consists of two segments. The first is the propagation of data from the ARF to the CRIB entry and execution. The second segment is the propagation of the result to the next ARF. There is a vast imbalance between the first and second segments, with the first segment being the critical one. In the right picture, the critical path is now separated into two segments, in which the first one is the data propagation from the sources (can be ARF or another CRIB entry) to the CRIB entry, while the second one is the execution itself. Completion bits are no longer latched in the ARF every cycle. Instead, they are propagated using transparent latches. Completion bits are latched in the ARF when the instructions in the CRIB finish executing, just as the data bits. The ALU latency is used as the critical path to determine the new cycle time. Our results show that we can double frequency with this pipelining technique. We found that the result can propagate as fast as the next four entries in the

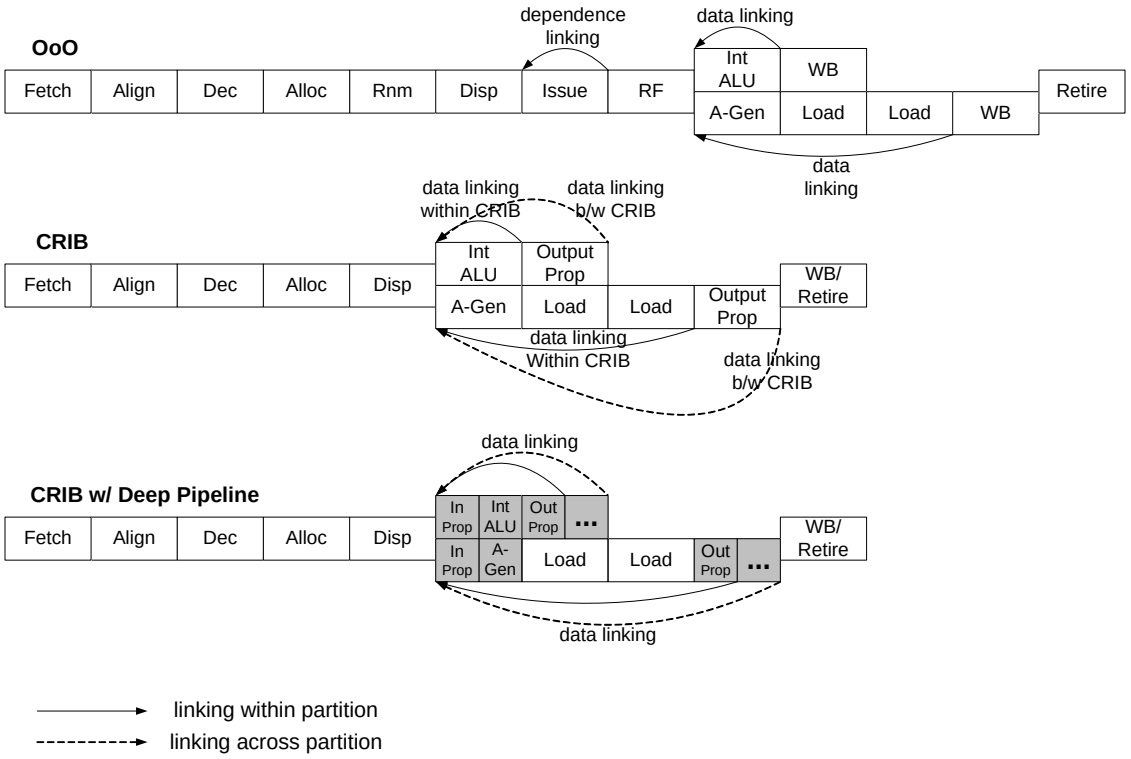


FIGURE 4-13. CRIB Pipeline Diagram. Shaded blocks are clocked at a 2x frequency

CRIB within a single clock period, reducing the length of the critical dependence path and improving CRIB partition utilization.

Figure 4-13 shows a pipeline diagram comparison between CRIB, CRIB with deep pipeline, and an out-of-order machine. It also shows how the pipeline changes as deeper pipelining is added. With the deep pipeline, shaded areas shown in Figure 4-13 are clocked at double frequency of the non-shaded area. The faster cycle is called a minor cycle while the rest of the processor uses major cycles. Input propagation, simple integer ALU execution, and output propagation each take one minor cycle. Depending on the distance between the producer and its dependents, the output propagation can take more than one minor cycle. Communication with the rest of the processor, dispatch, and writeback, is

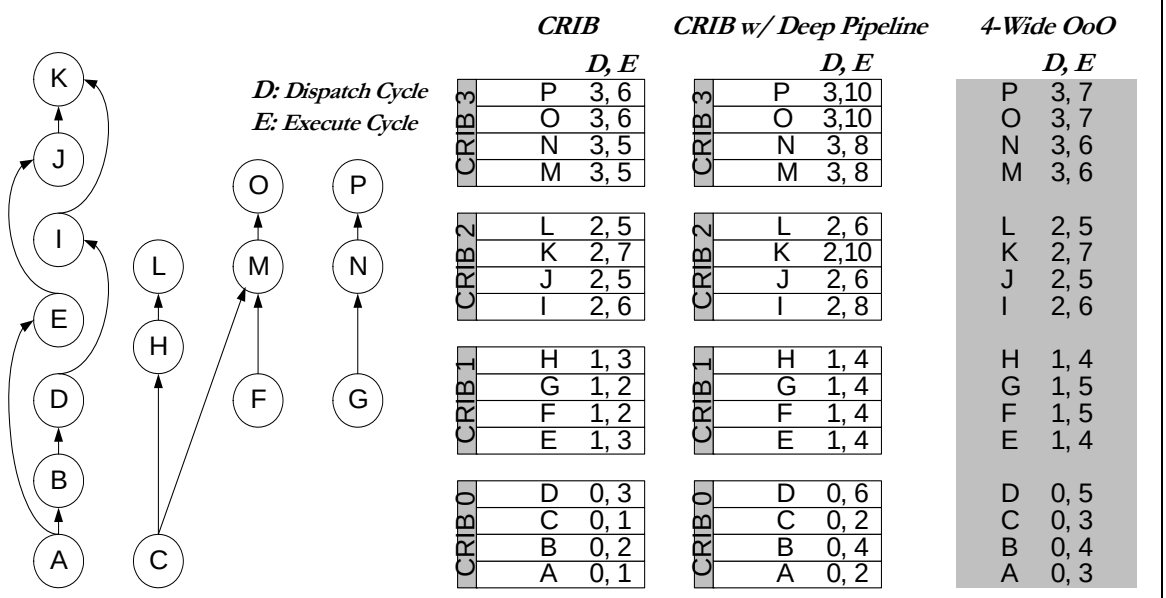


FIGURE 4-14. Execution Schedule Diagram. CRIB, CRIB w/ deep pipeline, and baseline out-of-order machine

done at major cycle boundaries.

Figure 4-14 illustrates when instructions are dispatched and executed in CRIB based on their dependencies. For comparison, instruction progress in a baseline out-of-order machine is also included. The example assumes the pipeline shown in Figure 4-13, hence the lag between execution cycle and dispatch cycle in the baseline. While CRIB with deep pipeline finishes with the highest cycle count, it is clocked at twice the frequency. Thus, it actually finishes after five major cycles, two cycles ahead of either the baseline or the shallow pipeline CRIB. This benefit will only occur for lengthy data dependency chains such as the one illustrated here, since it derives from compressing dependency chains into fewer cycles.

4.4 CRIB with Data Cache Optimization

One of CRIB's strengths is its ability to handle non-determinism in execution

latency, including data cache latencies. Modern out-of-order scheduling logic relies on speculative wake-up of load dependents. Non-deterministic delay from bank conflicts or line buffer misses create additional complexity to squash the pipeline and replay the instructions. However, CRIB does not separate dependence and data linking, thus it can tolerate the additional non-determinism introduced by many cache optimizations. In this thesis, we choose to pair CRIB with cache banking and line buffers to provide additional bandwidth and additional cache power saving.

4.4.1 Cache Banking

Cache banking is a technique commonly used in lower level caches to provide additional cache bandwidth without increasing the number of cache ports. However, cache banking is not commonly used in level-1 cache of out-of-order machines. While increasing the cache bandwidth, cache banking introduces an element of uncertainty in cache latency due to bank conflicts. In cache subbanking, the number of ports per bank is kept minimal, i.e. one port per bank. Bank conflicts occur whenever more than one load instruction being issued falls into the same bank. For bank conflicts, load instructions have to be squashed and replayed. However, since CRIB does not schedule dependent instructions speculatively, bank conflicts simply result in the load instruction being delayed.

4.4.2 Cache with Line Buffers

Line buffers are a technique proposed in the past to reduce cache energy by filtering cache accesses. Similar to cache banking, line buffers introduce additional variability to cache latency. A load instruction can possibly hit in the line buffer or not. Thus, the technique has not been used in current generation processors despite its simplicity.

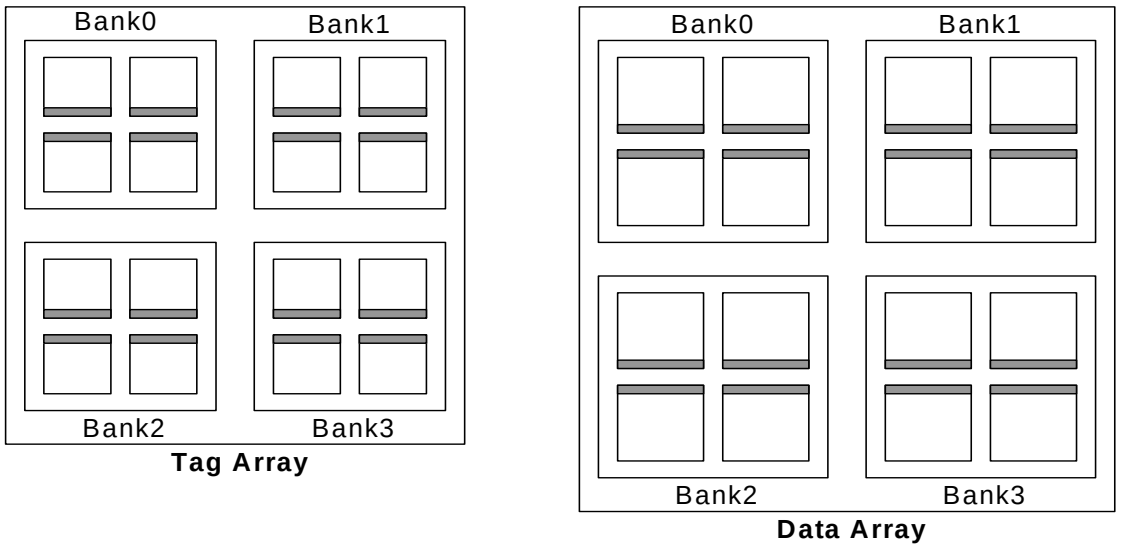


FIGURE 4-15. Cache Organization with Line Buffers. Line Buffers are shown in shaded area

While conceptually similar with prior line buffer work [79][80][27], our proposed line buffer is slightly different. All prior work on line buffer maintains line buffers as separate entities that can be thought of as a smaller write through cache that sits closer to the processor. Maintaining line buffers as separate entities is expensive since the whole cache block has to be routed out from the cache. All prior work also assumes a monolithic cache. While commonly used at the time, a monolithic cache has been replaced with a cache design with subarrays, where tag and data array are normally maintained separately within their own subarrays.

In this thesis, we adapt the line buffer concept into current generation cache design as shown in Figure 4-15. Figure 4-15 shows a four-banked cache with four subarrays in each bank. Tag and data are maintained separately in tag and data arrays. In our design, we utilize the pipeline latches that are already exist in the cache rather than maintaining line buffers as a separate entity. Cache pipeline latches used as line buffers are shown in shaded area in Figure 4-15.

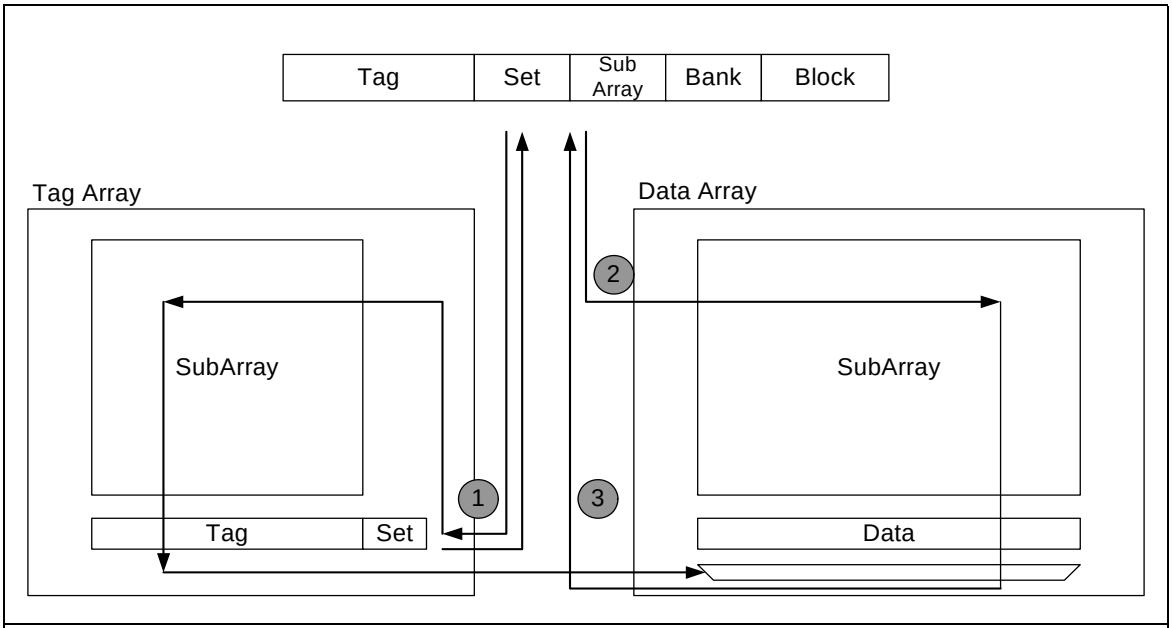


FIGURE 4-16. Line Buffer Operation. Minor cycle is annotated in the operation steps.

Line buffer operation is illustrated in Figure 4-16. The steps taken for line buffer access in each minor cycle are annotated in Figure 4-16. Those steps are described further below. Please note that the minor cycle is used for clarity. The access itself is done in a combinational manner.

Minor Cycle 1:

- *Send the effective address to tag array to access the line buffer associated with the subarray being accessed.*
- *If set hit on line buffer, return the hit/miss signal.*

  If set hit and tag miss, the cache access is done, start the L2 access.
- *If set miss, start the subarray access.*

Minor Cycle 2:

- *If set hit and tag hit, access the line buffer and return the data. Cache access is done in two minor cycles.*
- *If it is a set miss, start the data subarray access.*

Minor Cycle 3:

- *Tag comparison signal is available in data array*
- *Data is returned from the data array in case of hit*
- *Data is written into the load queue*

4.5 Chapter Summary

In this chapter, we explained the concept of CRIB. Rename, issue, and bypass are consolidated into one structure. Explicit register renaming is replaced with positional renaming within the CRIB. The CRIB entry itself is a connection matrix connecting logical register columns into an ALU. It also connects the ALU result back to the appropriate register column. Execution results are not latched within the CRIB. Instead, they are latched into the architected register file when all entries finish executing. While data propagation inside CRIB is done combinationally without latching, it is made synchronous by the propagation of the completion bit.

CRIB is scaled up by replicating the CRIB into a partitioned-CRIB. Partitioned CRIB is maintained in a circular fashion. Only the head of the CRIB has the committed state of the program. We also explain how CRIB handles different type of instructions in this chapter.To improve CRIB occupancy, deep pipelining is applied to CRIB, though only to the control bits. With deep pipelining, CRIB can be clocked twice the frequency.

Lastly, we pair CRIB with two cache optimization techniques, cache banking and line buffers. Cache banking is employed for additional bandwidth while line buffers are employed for power saving purpose.

Chapter 5

CRIB Design

This chapter describes the details of the CRIB design and presents modeling results. We chose to use the x86-64 architecture in designing the CRIB as it is the most relevant architecture today and in the future. However, we believe that CRIB design can be applied to any instruction set architecture.

The modeling of a CRIB entry is first described in detail in Section 5.1. Section 5.1 is separated further into Section 5.1.1 to explain the design of original CRIB entry and Section 5.1.2 to explain the changes necessary to implement deep-pipelined CRIB. Architected Register File modeling is illustrated in Section 5.2. CRIB modeling results are presented in Section 5.3. Last, low power data cache design to be combined with CRIB is explained in Section 5.4.

5.1 CRIB Entry

As explained in Chapter 4.1, each entry of CRIB consists of an ALU, input router, output router, and flip-flops for storing opcodes, input identifiers, and output identifiers, as shown in Figure 5-1. Input routing consists of tri-state buffers that are enabled by the source identifiers. These tri-state buffers drive the appropriate input to the ALU. For output routing, two-to-one muxes are used to choose whether to overwrite the register value to the next entry with its ALU result or to continue the previous value. The output identifier drives these muxes to overwrite the right register.

As illustrated in Figure 5-1, data portion of the CRIB is propagated without any

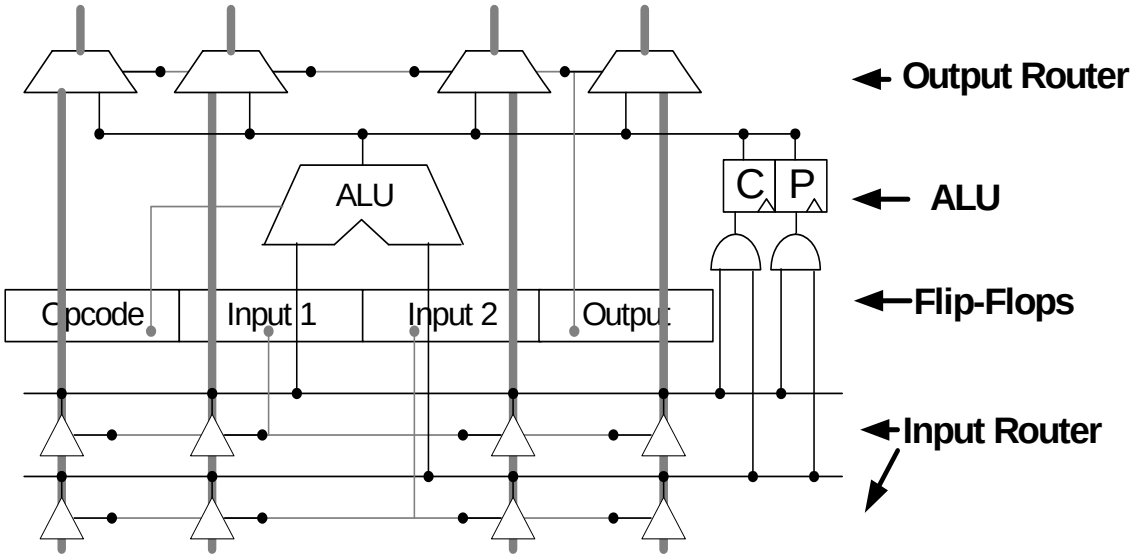


FIGURE 5-1. CRIB Entry. Shaded thick lines corresponds to 64-bit wire

latch. Outputs from the tri-state buffers are connected directly into the ALU; output from the ALU is connected directly into the two-to-one muxes. Data propagation is made synchronous by the control portion of the CRIB. The control bits of CRIB consist of completion bit and the poison bit. Completion bit, shown as ‘C’ latch in Figure 5-1, signals when the result of the CRIB entry is ready to be consumed by its dependents. The poison bit, shown as ‘P’ latch, signals whether the result produced is a speculative value due to runahead. Similar to the data portion, input identifiers route the completion bit and the poison bit from the register sources to the entry. The output identifier controls which register’s complete and poison bit to overwrite. These control bits are reset whenever a new instruction is put into the CRIB entry and are set based on the complete and poisoned bit of its sources.

Within a CRIB partition, multiple CRIB entries are placed adjacent to each other. The output router from one CRIB entry is directly connected to the input router of the next CRIB entry. Thus, consumers of a CRIB entry within a partition will be executed in the

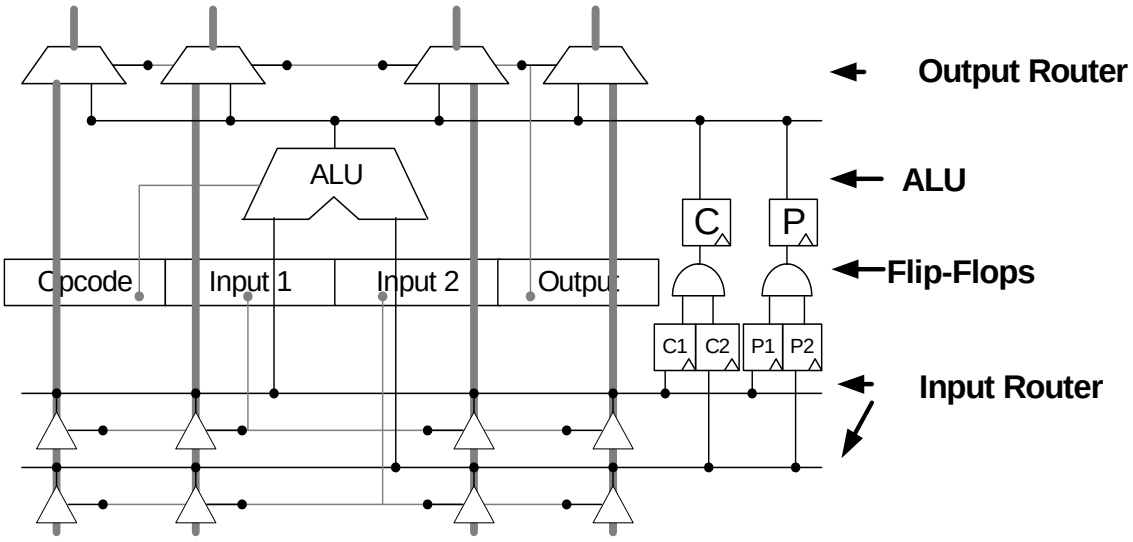


FIGURE 5-2. CRIB Entry with Deep Pipelining.

next cycle after the producer is executed; the completion bit of the consumers will be set the cycle after the completion bit of the producer is set.

Figure 5-1 shows the control path for CRIB entry without deep pipelining. With deep pipelining, two additional flip-flops are required for each of the control bit as shown in Figure 5-2. The input propagation and execution path is now broken down into execution path and propagation path. The propagation path is made synchronous with the completion bit latching. When both source operands' completion bits are set, the execution is done at the next cycle. At the end of the execution cycle, the completion bit of the entry is also set, which means that the entry is done executing. Poison bit propagation is also done in the same manner.

In a deep-pipelined CRIB, consumers within a partition are no longer executed in the cycle after the producer's result is available. Once the result is available, which is signifies by the completion bit being set, it will have to be propagated to the consumers' entries. The propagation cycles can take multiple cycles depending on the distance

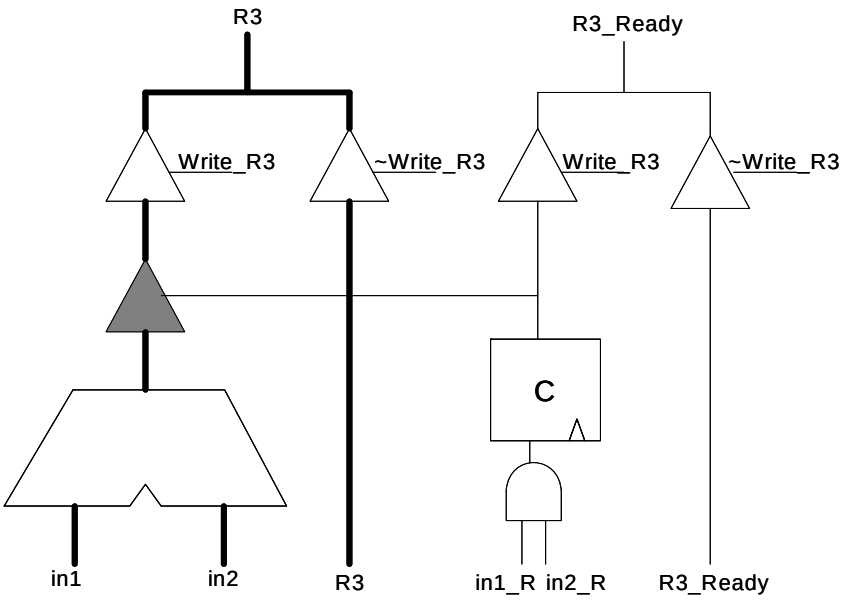


FIGURE 5-3. Output Router with Glitch Avoidance. Thick line represents 64-bit wires

between the producers and the consumers. Once the result reaches the consumer's entry, the input-completion bit is set. When both input completion bits are set, the entry will get executed and set its entry completion bit in the next cycle. Data propagation to consumers outside the partition is also done in a similar manner.

Since the datapath in CRIB uses minimum latching to minimize power consumed by flip-flops and clock tree, it is prone to glitching. Results can get propagated and consumed before they are fully ready. While this does not affect the correctness of the final consumers' output, it increases power consumption of the consumers' ALU. Transistors inside the consumers' ALU might switch back and forth a few times before settling to a final value. To avoid glitching power, another set of tri-state buffers is added to the output of the ALU. The tri-state is controlled by the completion bit and is enabled on the clock cycle following the execution. Thus, the output of the ALU is exposed to its consumers only when the result is stable. The output router with glitch avoidance is shown in

.

Figure 5-3. 81

The CRIB entry is the only storage that an instruction has during its execution, thus it needs to store all necessary information for each instruction, similar to an ROB entry. An integer CRIB entry has a total of 147 bits of storage. Out of the 147 bits, 145 bits will be written during dispatch, while complete and propagate bit will be set accordingly. A floating-point CRIB has 51 bits, 96 less bits than the integer counterpart since it does not need to have the instruction PC and the immediate value storage. The 147 storage bits in an integer CRIB consists of:

- Valid bit - 1 bit
- Instruction PC - 64 bits
- Opcode - 7 bits
- Left input source identifier - 20 bits
- Right input source identifier / immediate values - 32 bits
- Output identifier - 20 bits
- Load queue index - 1 bit
- Complete and poisoned bit - 2 bits

We chose to use the x86-64 architecture in this research as it is the most relevant architecture today and in the future. The x86-64 has sixteen general purpose register, 64 bits each, and a 32-bits EFLAGS register. For internal uop cracking, three temporary registers are added making the total integer register count twenty and the total number of bits 1280. For x87 floating point support, there are eight 80-bit floating point registers, one 16-bits control register, one 16-bits status register, and one 16-bits tag register. MMX technology requires eight MMX registers, 64 bits each, that are aliased to the x87 floating point

registers. SSE adds sixteen XMM registers, 128 bits each, and a 32 bits MXSCR. The instruction PC is also added to the integer register state. By adding the instruction PC into the register state, the CRIB partition no longer needs a write port to read out instruction state in the case of branch misprediction or exception. This specification results in 1312 bits and 2768 bits of spatial register state overlaid on the integer CRIB and floating point CRIB. Additional 2 x 20 bits (16 registers + 3 temporary registers + 1 flag) for integer CRIB completion and poison bits and additional 2 x 26 bits (8 x87 registers, + 1 control/status/tag + 16 XMM registers + 1 MXCSR) for floating-point CRIB completion and poison bits are added on top of the 1312 bits and 2768 bits stated above. Given this specification, Section 5.1.1 describes the design of the CRIB with an objective of reaching a competitive cycle time without sacrificing back-to-back execution of dependent instructions. Next, Section 5.1.2 explains how the design changes to accommodate deep pipelining.

5.1.1 CRIB

One restriction we use in our design is to match the cycle time of the baseline out-of-order machine described in Section 3.4. With this cycle time restriction, our design goal is to enable a back-to-back execution between a producer and a consumer within a CRIB. This parameter limits the number of CRIB entries that can be put into the execution array as the output of the producer needs to travel through the output routers of each intervening entry before reaching its consumer. Related to this parameter, the output of an entry should also be able to reach the architected register file in one cycle so that writeback can occur in the cycle following execution of the last entry.

For this cycle time goal, we choose the larger latency between the wakeup-select

Table 5-1: Delay, Energy, and Area for CRIB.

		Integer CRIB			Floating-Point CRIB		
		Delay (ns)	Energy (pJ)	Area (um^2)	Delay (ns)	Energy (pJ)	Area (um^2)
Input Router	Driver	0.08	1.43	10240	0.08	2.01	18576
	Tri-State	0.07			0.07		
ALU	ALU	0.56	5.84	10214	0.55	11.68	20428
	Tri-State	0.05			0.05		
Output Router	Driver	0.08	0.75	4561	0.08	0.86	8395
	Mux	0.05			0.05		
Storage	Delay	0.14	0.69	1164	0.14	0.60	404
	Setup Time	0.11			0.11		
Area of One CRIB Entry				26021			48150
Area of Four-Entry CRIB				104716			191212

loop in the reservation station and the ALU-bypass loop in the functional units. The wakeup-select loop latency is 0.90 ns and ALU-bypass loop is 0.69 ns. Thus the critical path latency is 0.90 ns before adding flip-flop overhead. Flip-flop delay is 0.14 ns and flip-flop setup time is 0.11 ns, setting a baseline cycle time of 1.15 ns. While this latency is clearly not competitive with current generation full-custom designs, we believe that relative comparisons are still meaningful because both the baseline and the proposed design use the same design methodology.

We categorize the combinational delay of our proposed CRIB entry into four caregories: the input router, the ALU, the output router, and the flip-flops. For the integer CRIB, the input and output router has to route 20 registers . Correspondingly, the floating-point CRIB has to route 28 registers. Delay, energy, and area estimates are shown in Table 5-1.

The delay of the input and output router is separated into the driver delay and the tri-state delay. The driver is a series of inverters to strengthen the input/output identifiers stored in the flip-flops to drive the tri-states/muxes to choose its input or overwrite the output. The driver delay only affects the delay when the instructions are first moved into the

CRIB and it is often hidden. ALU delay is separated into the ALU combinational delay and the gitch avoidance tri-state delay. The worst combinational delay for a CRIB partition occurs when all entries are simple ALU instructions that are independent of each other. They are all executing on the very next cycle after being dispatched and they are all writing back the cycle after. The worst delay during the execution cycle happens for the fourth entry where its input has to travel through the output router of the first three entries before being executed in its ALU. So the worst combinational delay for the execution cycle is 0.08 (driver) + 3 x 0.05 (three output router) + 0.07 (input router) + 0.56 (ALU), equal to 0.85 ns, comparable to the baseline combinational delay. For the write-back cycle, the worst combinational delay occurs for the first entry where the result has to travel all the way to the ARF. Its delay is 0.05 (glitch avoidance tri-state) + 4 x 0.05 (four output router), equal to 0.25 ns. So the worst combinational delay of a four-entry CRIB is 0.85 ns, slightly less than the worst combinational delay of the baseline's 36-entry reservation station delay of 0.90 ns.

Energy is calculated using the assumption that only two out of the 20 tri-states in the input routers are switching as each input router only reads one out of sixteen registers. A similar assumption is applied to the output router energy calculation. For the storage bits, we assume 147 bits for integer CRIB and 51 bits for the floating-point counterpart. The storage energy is the energy consumed when a group of instructions is inserted into the CRIB, assuming a 50% switching factor.

We do not have detailed delay, energy, and area estimates for the MMX/SSE integer vector unit that is integrated into each floating-point CRIB entry. However, we believe that the delay should be approximately the same as the integer ALU because the short vec-

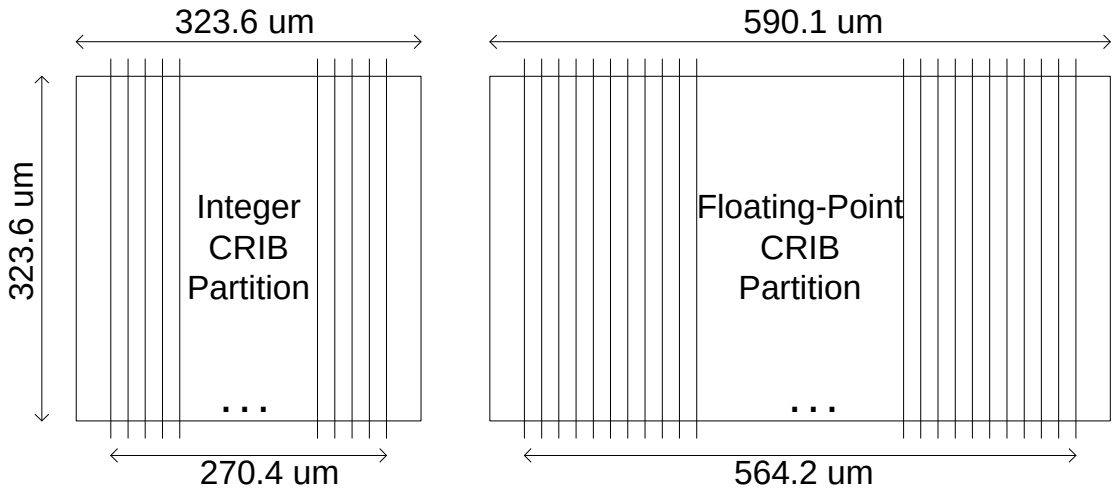


FIGURE 5-4. Proposed CRIB Partition Floor Plan.

tor arithmetic operations do not have to propagate a full 64-bit carry. The area and energy is estimated as twice the integer unit because the XMM registers are 128 bits wide, vs 64 bits in the integer ALU.

We also consider the dimensions of the CRIB with 1312 + 20 + 20 (integer) and 2768 + 26 + 26 (FP) register path wires going across. We choose metal layer 4 for the register columns. Following the layout rule specified by the library, metal layer 4 requires a minimum width of 0.1 um and a minimum spacing of 0.1 um. Thus, the total width of the register columns is 270.4 um and 564.2 um. The area of four-entry integer and floating-point CRIB are 104716 um^2 and 191212 um^2. By taking the square-root of the integer CRIB partition as the height and the width of the integer CRIB partition, integer CRIB partition has 323.6 um as its height and its width as shown in Figure 5-4. Maintaining the same height as the integer CRIB partition, floating-point CRIB partition has a width of 590.1 um Hence, the CRIB area should not be wire-dominated, despite the seemingly large number of wiring channels that span it, even when all wires are in a single metal layer.

While the delay calculation above already takes cell capacitance into account, it has not calculated the time for the signal to travel through the wire itself. The wire delay calculation using distributed RC model [40], $1/2R_{wire}C_{wire}$, for 564.2 um x 0.1 um metal layer 4 is 0.052 ps. This calculation is done with R_{wire} equal to 0.14 ohms per square distance and C_{wire} equal to 0.000232 pF per square distance. Thus, the wire delay can be considered negligible in the cycle time delay calculation. Although register columns appear to be long global wires, they are in fact short point-to-point links, as they are repeated at each CRIB entry output router.

5.1.2 CRIB with Deep Pipelining

With deep pipelining, the cycle time restriction to enable back-to-back execution within a CRIB partition is no longer relevant. In deep pipelined CRIB, the control bits are no longer being latched at the architected register file every cycle. Thus, the producer-consumer data propagation delay is no longer determined by the number of CRIB partitions separating the two. Instead, data propagation delay is determined by the number of CRIB entries between the producer and consumer.

As mentioned earlier, the data propagation path in the deep-pipelined CRIB is separated into the execution path and the propagation path. As the path becomes shorter, deep-pipelined CRIB can be clocked at a faster rate. For deep-pipelined CRIB, the cycle time latency is calculated using the following equations:

cycle time = max(data latency, control latency).....(1)

execution_latency = max(ALU latency, ready latency) .....(2)

propagation latency = max(data latency, control latency).....(3)

The execution path can be separated into ALU latency and ready latency, while the propa-

gation path can be separated into data and control propagation. Propagation latency for equation (3) depends on how many CRIB entries the propagation spans. Thus, we use execution path (2) to determine the cycle time. Ready latency is calculated as an AND gate delay plus FF setup and delay time. ALU latency is simply the ALU latency of 0.56 ns because the datapath is not latched. Using a cycle time of 0.56 ns, the number of entries that completion bit can propagate is calculated to be four entries per the following calculation:

$$0.56 = 0.25\ (FF\ overhead) + x * 0.05\ (output\ router\ delay) + 0.04\ (transparent\ FF\ delay) + 0.07\ (input\ router\ delay)$$

Thus, with a cycle time of 0.56 ns, the data and ready bit can propagate to the next four entries. Using this cycle time, the CRIB is clocked at twice the rate of the baseline machine, which is 1.15 ns.

5.2 Architected Register File

Transparent flip-flops [38][43] are used for the data portion of the architected register file to allow data values to travel from one CRIB partition to the next without being latched. However, regular flip-flop is still used for completion bits and poison bits of the architected register file. Table 5-2 shows the delay characteristic of both flip-flop as characterized by Hill et al. [38]. Reference [38] shows the energy characterization differentiated by the fact whether the input signal is rising or falling. For simplicity, Table 5-2 shows the average of the two. For transparent flip-flops, the switching energy is differentiated into two terms. The first one is when the switch occurs before the clock is applied and the second term is after the clock is applied. The area is taken from our TSMC library for

Table 5-2: Delay Characteristic of Regular and Transparent Flip-Flop.

	Latency (ns)			Energy (fJ)		Area (um^2)
	t_{setup}	clock to Q	transparent	Switch	No-Switch	
Regular Flip-Flop	0.11	0.14	N/A	5.50	0.72	7.92
Transparent Flip-Flop	0.11	0.14	0.22	4.78 + 0.72	0.72	7.92

Table 5-3: Energy and Area of Architected Register File in one CRIB partition.

	Energy (pJ)	Area (um^2)
Integer ARF	0.199	21415.68
Floating-Point ARF	0.398	44668.80

regular flip-flops. Given the fact that transparent flip-flops do not add anymore transistors into a flip-flop design, except for adding more control signals, we assume they both have the same area.

Every CRIB partition has two copies of the architected register file (ARF): the active copy and the checkpointed copy. Each of these copies has 1312 and 2768 transparent flip-flops for integer and floating-point ARF. An additional 40 and 52 regular flip-flops are added for completion and poison bits for integer and floating-point ARF. Thus, the total number of flip-flops for the integer ARF is 1352 x 2 = 2704 and the total number of flip-flops for floating-point ARF is 2820 x 2 = 5640 for each partition. Table 5-3 shows the energy per write and area for integer and floating-point architected register file. Write energy is calculated using 50% switching factor; it assumes perfect clock gating, only the register being written is clocked. Read energy is considered free since there is no port associated with the architected register file.

5.3 Modeling Results

We use the area of the baseline2 machine to determine the number of CRIB partitions that we can have. Our modeling results show that four four-entry CRIB partitions (4x4) have comparable area with baseline2, which is an Intel Nehalem like out-of-order

Table 5-4: Energy and Area of Baseline Machine.

	Four-Partitioned CRIB		Baseline1 OoO Machine		Baseline2 OoO Machine	
	Energy (pJ)	Area (mm^2)	Energy (pJ)	Area (mm^2)	Energy (pJ)	Area (mm^2)
Integer Register Alias Table	-	-	3.29	0.058	3.29	0.058
Floating-Point Register Alias Table	-	-	3.25	0.062	3.25	0.062
Reorder Buffer	-	-	9.16	0.100	11.83	0.302
Reservation Station	-	-	10.90	0.071	13.71	0.322
Immediate RAM	-	-	3.53	0.025	3.75	0.056
Integer Register File	0.199	0.086	8.04	0.151	9.82	0.282
Floating-Point Register File	0.398	0.179	17.03	0.308	18.50	0.481
INT CRIB / INT ALU + bypass	13.96	0.419	7.73	0.050	7.73	0.075
FP CRIB / FP ALU + bypass	21.17	0.764	12.63	0.026	12.63	0.026
Load Queue	2.32	0.015	2.32	0.015	13.33	0.204
Store Queue	1.16	0.008	1.16	0.008	10.56	0.093
Level-1 Data Cache	51.06	0.546	51.06	0.546	51.06	0.546
Total Area		2.017		1.420		2.507

processor. We model impacted structures in both baseline1 and baseline2 as explained in Section 3.4. Machine configuration in Table 3-1 is used to determine the dimension of each structure. Energy and area comparisons among these structures are shown in Table 5-4. The energy listed is energy per access while the area shown is the total area for each structure. We assume that complex floating-point units remain the same without any modification so we do not model those structures.

The area and energy shown in Table 5-4 is derived from the area and energy of CRIB entries shown in Table 5-1 and the area and energy of architected register file shown in Table 5-3. The ALU energy for CRIB includes the energy to insert instruction into the CRIB, input router energy to read input operands, the ALU itself, and eight times output router energy. This is assuming that on average each destination result has a lifetime of eight instructions ahead before being overwritten. Load queue and store queue energy for CRIB is the same as the load queue and store queue energy for baseline1 machine since both have the same queue size. The level-1 data cache is modeled as a single banked cache using CACTI.

Table 5-5: Banking Results Comparison for 64KB, 64B, Direct Mapped Cache.

	Latency (ns)	Energy (pJ)	Area (mm^2)
1-Banked	1.24	51.06	0.546
4-Banked	1.23	42.39	0.717

Table 5-4 shows that CRIB has an area of 2.017 mm^2, 19% smaller than baseline2 and 42% larger than the low-performance baseline1. Table 5-4 also shows that several major structures are eliminated in CRIB. The area estimation can also be used to estimate the leakage power of CRIB design, as it is mostly proportional with area. Besides removing major structures, CRIB also removes many pipeline latches. We estimate a removal of 1420 pipeline latches, resulting in additional area saving of 0.017 mm^2 and energy saving of 3.935 pJ per cycle. The power reduction of the simpler clock-tree needed for the CRIB design is harder to quantify, as there are fewer pipeline latches that need to be clocked.

5.4 Cache Optimization

As explained in Section 4.4, CRIB allows the usage of cache optimization without suffering the side effect of latency non-determinism. We use cache banking to provide additional bandwidth for the deeply pipelined CRIB and line buffers to save additional power.

5.4.1 Cache Banking

Cache banking increases the bandwidth of a cache without increasing the number of ports nor pipelining it deeper. Table 5-5 shows the cycle time, latency, energy per access, and area of a 1-banked cache and a 4-banked cache. Both 1-banked cache and 4-banked cache have similar latency. The resulting area of a 4-banked cache is slightly larger than 1-banked cache, 0.717 mm^2 compared to 0.546 mm^2. The energy per access is

.

Table 5-6: Delay and Energy Components for 64KB, 64B, DM, 4Bank Data Cache with Line Buffer.

Tag Array			Data Array		
	Delay (ns)	Energy (pJ)		Delay (ns)	Energy (pJ)
Routing In	0.11	0.90	Routing In	0.11	3.97
Row Decoder	0.23	0.46	Row Decoder	0.33	1.01
Bitline	0.04	0.87	Bitline	0.20	8.94
Sense Amplifier	0.01	1.94	Sense Amplifier	0.01	1.94
Line Buffers R / W	0.00 / 0.15	0.00 / 0.64	Line Buffers	0.00 / 0.15	0.00 / 6.37
Comparator	0.10	0.59	Subarray Output Driver	0.16	5.75
Subarray Output Driver	0.09	0.04	Routing Out	0.11	8.49
Routing Out	0.11	0.09	Precharge	N/A	2.14
Precharge	N/A	2.14			
Line Buffer Hit	0.41	1.62	Line Buffer Hit	0.38	18.21
Line Buffer Miss, Cache Hit	0.84	7.67	Line Buffer Miss, Cache Hit	1.07	38.61

slightly smaller, 42.39 pJ compared to 51.06 pJ.

With cache banking, load instructions can now be issued every minor cycle (up to two load instructions issued per major cycle) rather than one load instruction every major cycle. Increasing cache bandwidth also requires an increase in store queue associative search bandwidth. With only a 4-entry store queue, pipelining the store queue to enable an access every minor cycle should not be difficult.

5.4.2 Line Buffers

Line buffers give the advantage of performance and power at the same time. On a line buffer hit, it only takes 2 minor cycle to do the cache access while it takes 3 minor cycles in the case of a line buffer miss. Table 5-6 shows the delay and energy components for a 4-banked 4-way 64KB data cache with 64 byte blocks. Total energy spent when the access hits in the line buffer is 19.83 pJ, roughly 43% of the 42.39 pJ of energy spent on a full access. When an access hits in the line buffer but turns out to be a miss (no tag match), the energy spent is lower, 7.87 pJ, because the line buffer in the data array is not even accessed. In our design, line buffers are implemented using latches. The delay and energy

shown for the line buffers are divided into two categories: read access during line buffer hit and write access during line buffer miss. Since the line buffer are implemented using latches, the energy and delay for a read is practically zero (no switching). The non-zero energy and delay shown in Table 5-6 is for writing new values into the latches, which happens in case of a line buffer miss.

The total delay in the case of a line buffer hit is 0.79 ns, which nicely fits into two minor cycles. The total delay in the case of line buffer miss but cache hit is 1.48 ns, barely a fit into three minor cycles. After adding the FF setup and delay into the 1.48 ns to write the data back into the load queue, it becomes 1.73 ns. The delay is slightly larger than three minor cycles and thus considered as four minor cycles. These delays are calculated using the following equations:

Line Buffer Hit Latency =

Tag Routing In + Tag Comparator + Tag Subarray Output Driver +
Tag Routing Out + Data Routing In + Data Subarray Output Driver +
Data Routing Out

Line Buffer Miss Latency =

Tag Routing In + Tag Comparator + max (
(Tag Row Decoder + Tag Bitline + Tag Sense Amp + Tag Line Buffer +
Tag Comparator + Tag Subarray Output Driver + Tag Routing Out +
Data Routing In),
(Tag Subarray Output Driver + Tag Routing Out +
Data Routing In + Data Row Decoder +
Data Bitline + Data Sense Amp +Data Line Buffer)) +

5.5 Chapter Summary

In this chapter we describe the implementation of CRIB in detail. X86-64 architecture is assumed for CRIB implementation. CRIB's delay, energy, and area are analyzed. Using the cycle time limitation of baseline2, four-entry CRIB partitions are chosen. Using area limitation of baseline2, the CRIB is configured to have four four-entry partitions. The latency, energy, and area of the cache with banking with line buffers are also analyzed. CRIB has comparable cycle with baseline2. In the final area comparison, CRIB's area with 4-banked cache is 2.188 mm^2 compared to baseline2 area of 2.507 mm^2 .

The next chapter presents a detailed performance and energy evaluation of CRIB implementation. Energy analysis done in this chapter is used to calculate the total energy consumption for each benchmark. We will also perform sensitivity studies and detail benchmark analysis to help explain our results.

Chapter 6

Experimental Evaluation of CRIB

This chapter presents a detailed performance and energy evaluation of our implementation of CRIB. It begins with sensitivity studies in Section 6.1 to see if our design configuration explained in Section 5.3 is an optimal configuration. We first compare the IPC of CRIB with different number of CRIB entries to see the sensitivity of CRIB performance in response to window size. Using the same number of CRIB entries, we then compare different numbers of hops for deeply pipelined CRIB. A performance and energy comparison with our two baseline machines is provided next in Section 6.2. The performance comparison considers different aspect of CRIB that could possibly have impact in the performance and quantify them separately. Energy comparison is broken down into different structures to help understand where the energy saving comes from. The impact of cache banking and cache with line buffers is discussed in Section 6.3 . Throughout this chapter we also present characterization data to provide additional insight into our results. Detail per-benchmark analysis for some of the outlier benchmarks are explained in further details in Section 6.4.

Details regarding the baseline machine configuration and CRIB configuration used in this evaluation can be found in Section 3.2 and Section 5.3. We selected two different baseline machines in our evaluation. Baseline1 machine is an out-of-order machines with window size similar to the CRIB configuration being used. Baseline1 mainly serves as sanity check for our CRIB performance comparison. Baseline2 machine represents current generation aggressive out-of-order machine with large window, modeled after Intel

Nehalem core [76]. Unless otherwise specified, ‘CRIB’ in figures refers to CRIB with deep pipelining.

6.1 Sensitivity Studies

In this chapter we present two sensitivity studies. The study discussed in Section 6.1.1 studies performance sensitivity to the number of CRIB entries. Maintaining the same number of CRIB entries, we study different CRIB partition configurations in Section 6.1.2.

6.1.1 Performance Sensitivity to the Number of CRIB Entries

While the area limitation has been used to determine how many CRIB entries to be used in our design, we conduct a sensitivity study to see if sixteen CRIB entries is the most cost effective choice in extracting instruction level parallelism. Figure 6-1 shows the result of the sensitivity study for both SPEC INT and SPEC FP benchmarks from SPEC CPU 2006 suite. CRIB designs with various number of entries (4, 8, 16, 32, 64, and 128 entries) are studied. All configuration uses four-entry partitioned CRIB, deeply pipelined with 4-hops per minor cycle. The IPC shown in Figure 6-1 is normalized to the IPC of CRIB with four entries.

From Figure 6-1 it can be seen that the 16 entry configuration is enough to extract most of the instruction level parallelism for many of the benchmarks. This trend is especially true for SPEC integer benchmarks. For many of them the rate of IPC increase slows down after 16 entries, except for bzip2, omnetpp, and perlbench for which 32 entries configuration seems to be the most optimal for instruction level parallelism extraction. For SPEC FP benchmarks, 16 entries seems to be optimal for calculix, GemsFDTD, and

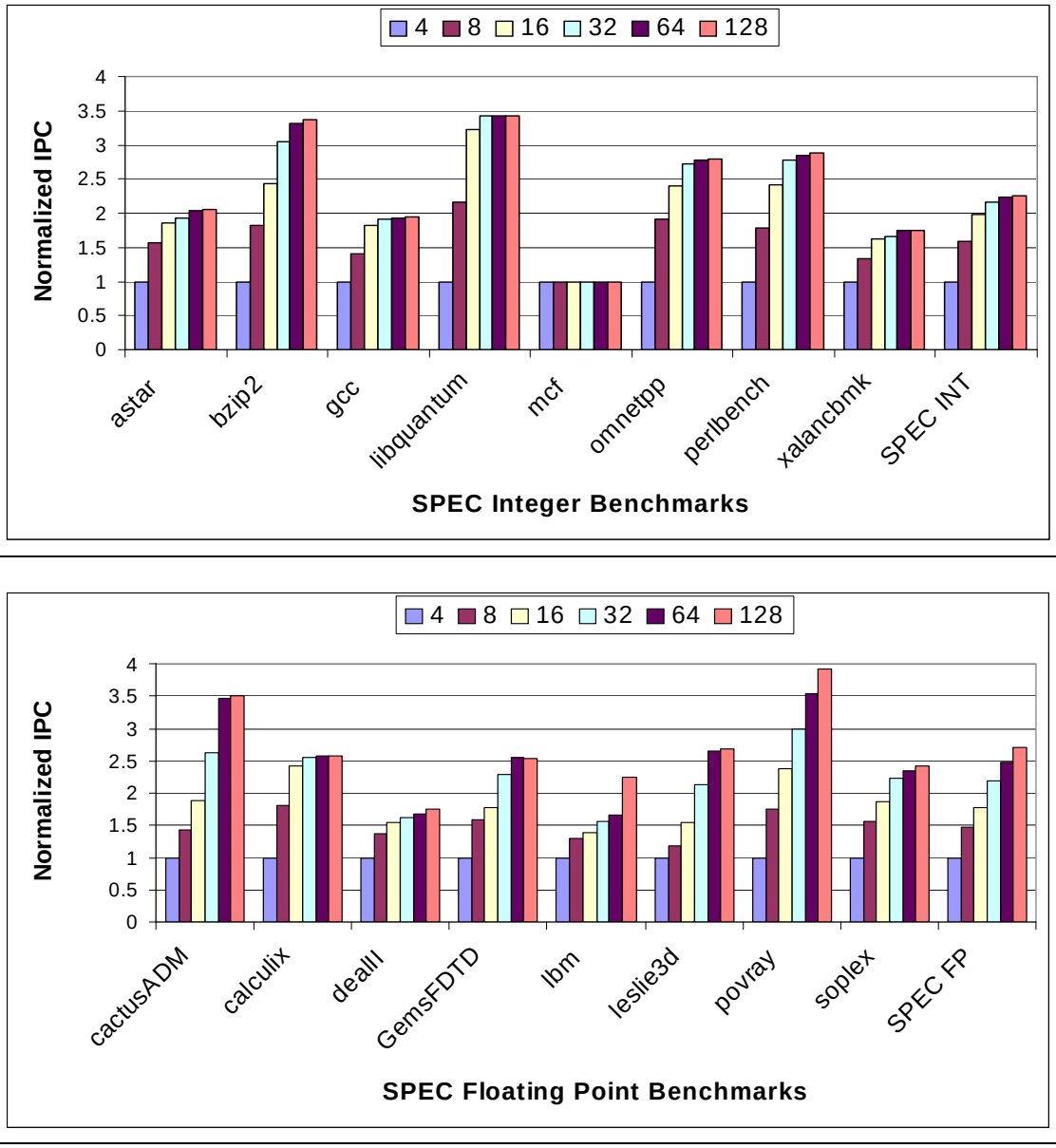
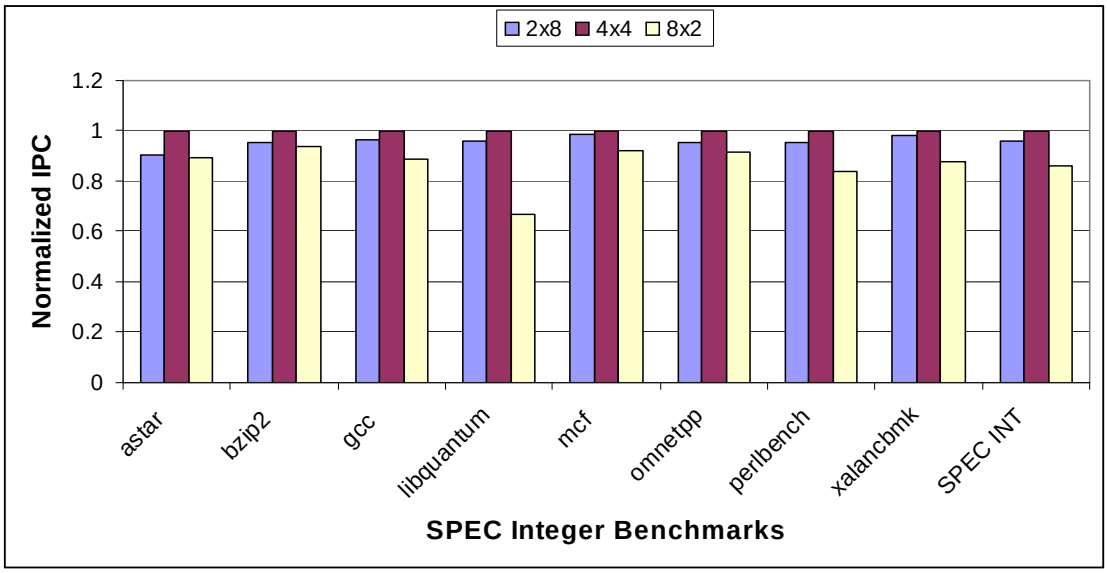


FIGURE 6-1. Number of Entries Sensitivity Study.

leslie3d. For others, additional entries seem to result in discovering further independent instructions that lead to increased in IPC. In general, integer benchmarks have longer dependency chains than the floating point benchmarks. Many of these floating-point benchmarks have a small independent loop that is repeated for millions of dynamic instructions. Increasing the number of entries in CRIB means increasing the out-of-order window where more independent instructions from the next loop iteration can be found.



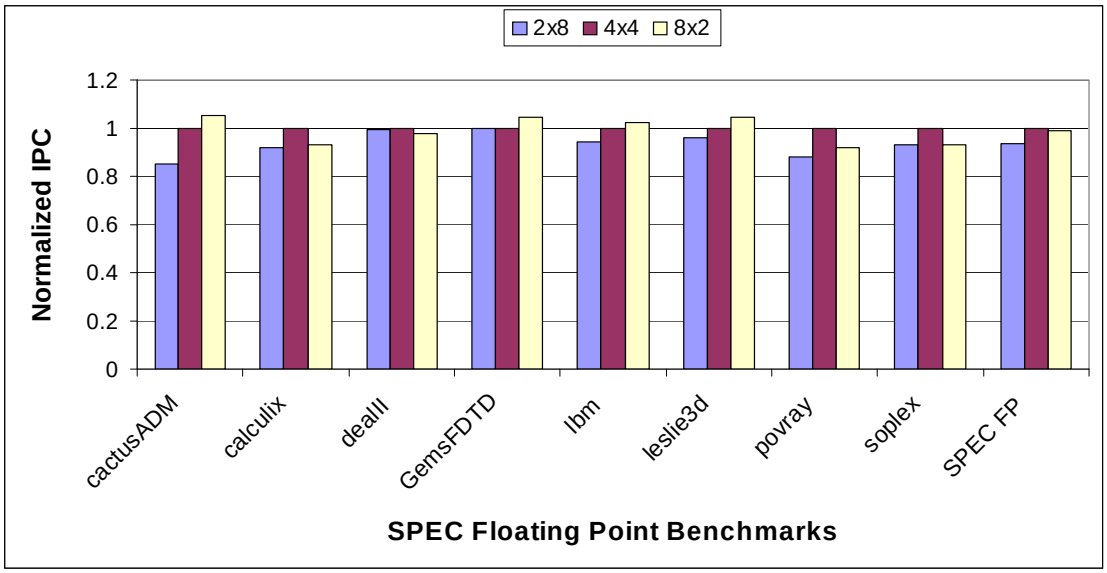


FIGURE 6-2. Partition Configuration Sensitivity Study.

An interesting case is the lbm benchmark where the increase of number of entries does not result in significant IPC increase until it hits 128 entries. The reason is because the size of the loop being repeated during execution is more than 64 instructions. Thus significant increase in IPC only happens when the window size is larger than the loop, allowing pipe-lining of multiple iterations of the loop execution.

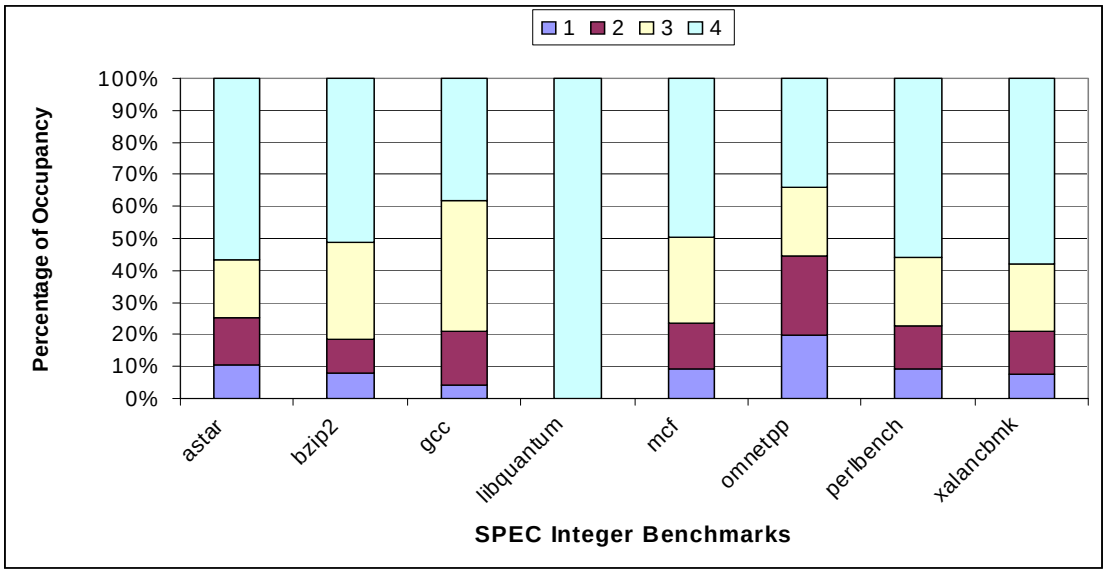
6.1.2 Performance Sensitivity to Different CRIB Partitioning

In order to see if we have chosen the most optimal CRIB partitioning configuration given sixteen CRIB entries, we examine performance of different partitioning approaches: two partitions of eight-entry CRIBs, four partitions of four-entry CRIBs, and eight partitions of two-entry CRIBs. A lower number of entries per partition has the advantage of faster instruction turn around. However, the architected register file inserted between every partition is not free. Besides the area and energy cost, it also introduces some latency thus reducing the number of entries that a destination register can propagate through. Based on our modeling described in Section 5.1 and Section 5.2, four partitions of four-entry CRIBs can propagate a result to the next four entries per minor cycle. Two partitions of eight-entry CRIB can only propagate to the next three entries per minor cycle, while eight partition of two-entry CRIB can propagate to the next five entries in one minor cycle. IPC is normalized to the IPC of four four-entry CRIB partitions.

Figure 6-2 shows that four 4-entry CRIB has the highest IPC on average. For integer benchmarks, shorter propagation distance (small partition) hurts more than slower turn-around rate (large partition). But for some floating-point benchmarks, shorter propagation distance does not negatively impact IPC as much. In fact, the benefit of faster turn-around rate of small partition increases IPC despite the shorter propagation distance, as seen for cactusADM and lbm. This is caused by short dependency chains commonly associated with floating-point benchmarks. From the study, we conclude that four 4-entry CRIB partitions is the most cost-effective configuration.

6.1.3 CRIB Occupancy

There are limitations imposed in dispatching instructions into the CRIB. Upon



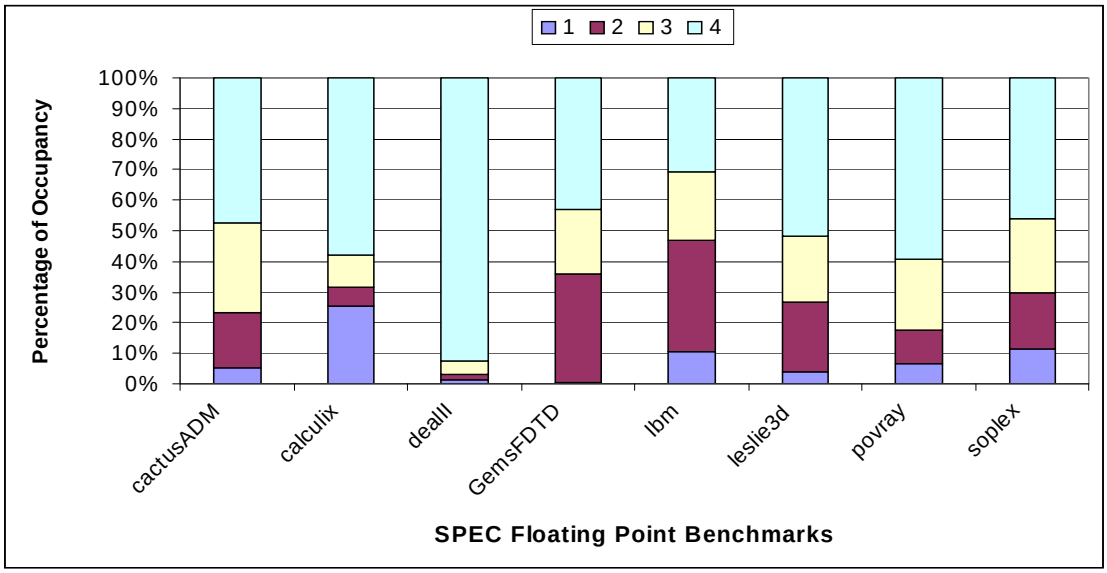


FIGURE 6-3. CRIB Occupancy.

reaching the limitation, the dispatch group is terminated and the next instruction is forced into the next group. Those limitations are 2 load instructions, 1 store instruction, 1 branch instruction, 1 complex integer instruction, 2 floating-point instructions, 2 complex SIMD instructions. The goal of the study is to understand if the limitation severely affects CRIB's occupancy.

Figure 6-3 shows the percentage of time when a four-entry CRIB partition is occu-

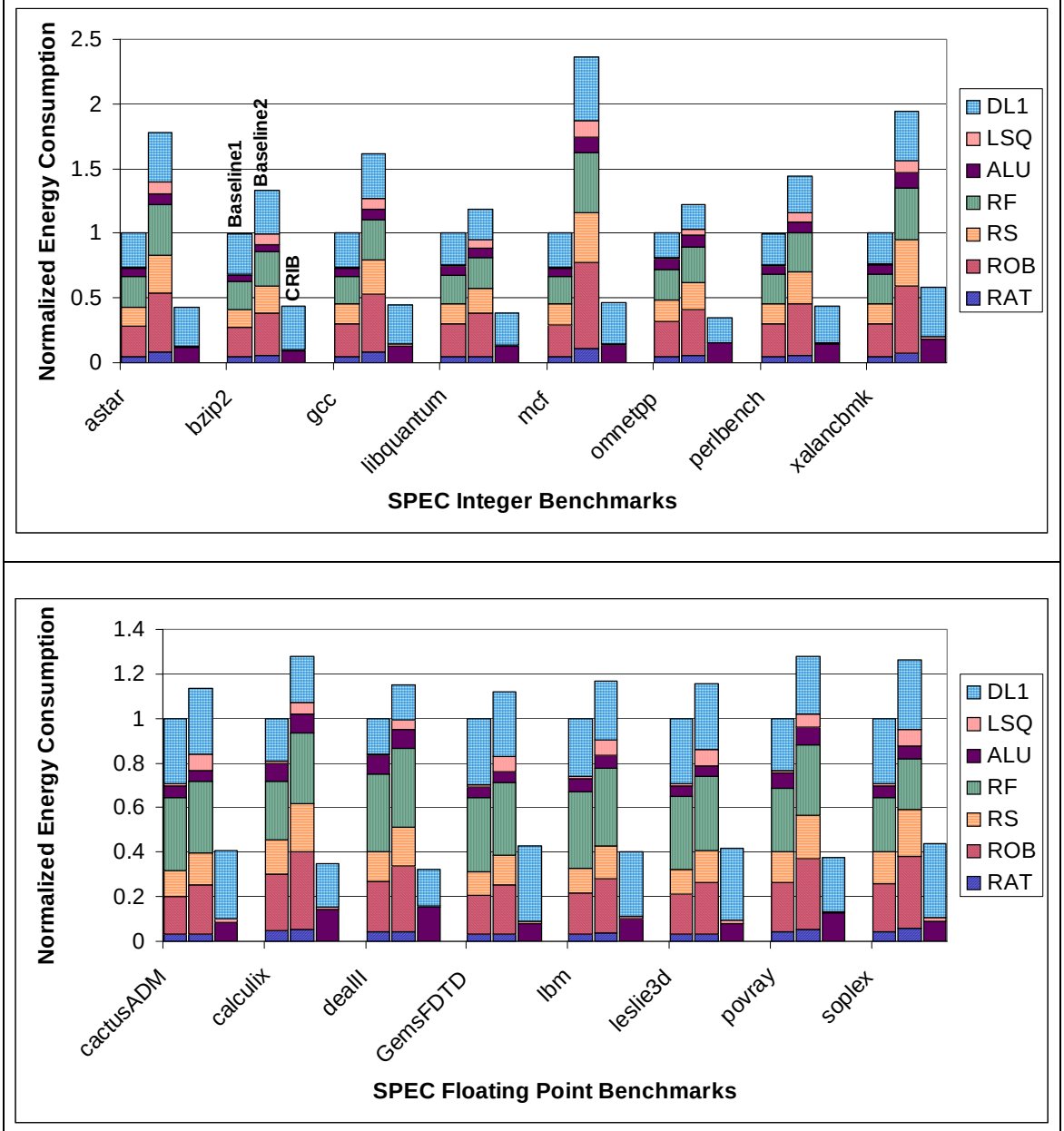


FIGURE 6-4. Energy Comparison.

pied by one, two, three, or four instructions. It can be seen that roughly 75% of the time, a CRIB partition is occupied by three or more instructions. Only a few benchmarks such as GemsFDTD, lbm, and omnetpp have low occupancy. Thus, we conclude that our limitation should not limit CRIB's performance significantly.

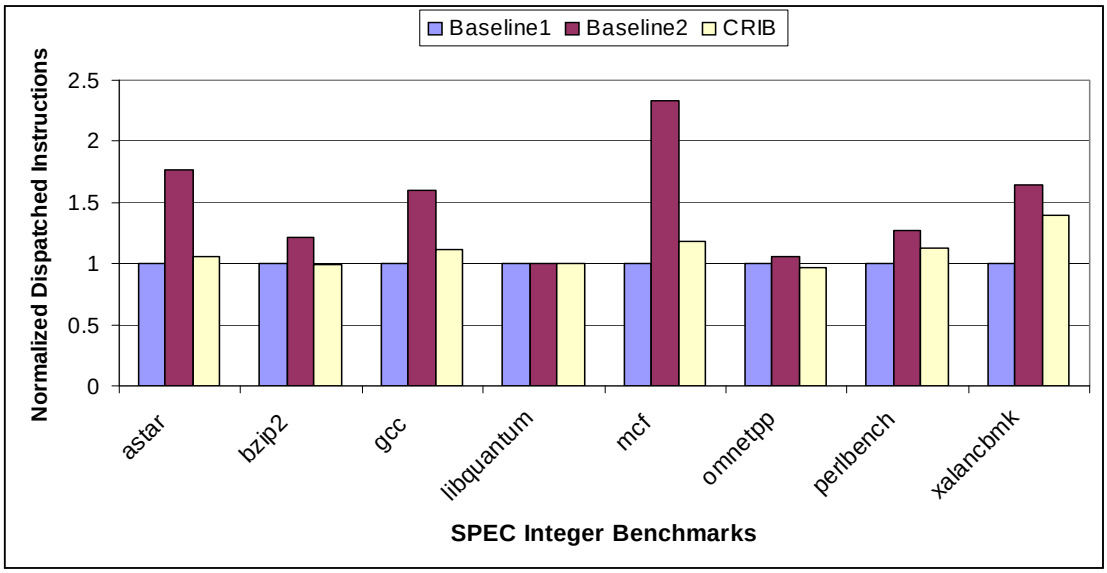
6.2 Energy and Performance Comparison

In this Section 6.2, we present an energy and performance comparison between CRIB and the two baseline machines.

6.2.1 Energy Comparison

Figure 6-4 shows energy consumption comparison between baseline1, baseline2 and CRIB. It is normalized to baseline1 energy consumption. On average, CRIB consumes 40% and 44% of baseline1 energy for integer and floating-point benchmarks. Compared to baseline2, CRIB consumes 28% and 33% of baseline2 energy. Most of the energy savings comes from removal of various structures such as RAT, ROB, RS, and RF. CRIB consumes more ALU energy than the baseline machines because ALU energy consumption of CRIB includes insertion energy, input routing, and output routing. The load store queue energy consumption of CRIB and baseline1 are roughly the same, and is about 15% of baseline2 energy due to the difference in size.

While we do not have an energy model for the front-end unit, we try to estimate it using the number of dispatched instructions. Because we do not change the front-end part of the processor, the relative number of dispatched instructions can be used to compare front-end energy between baseline1, baseline2, and CRIB. Figure 6-5 shows the normalized number of dispatched instructions among the three cores. In general, the number of dispatched instructions increased from baseline1 to baseline2 as the instruction window is enlarged. The more pipeline flushes induced by a benchmark, the larger the number of instructions to dispatch as the instruction window grows. For example, in a processor with a 24-entry reorder buffer, at most 24 instructions have to be redispatched during pipeline flush, while in a processor with a 128-entry reorder buffer, up to 128 instructions will have



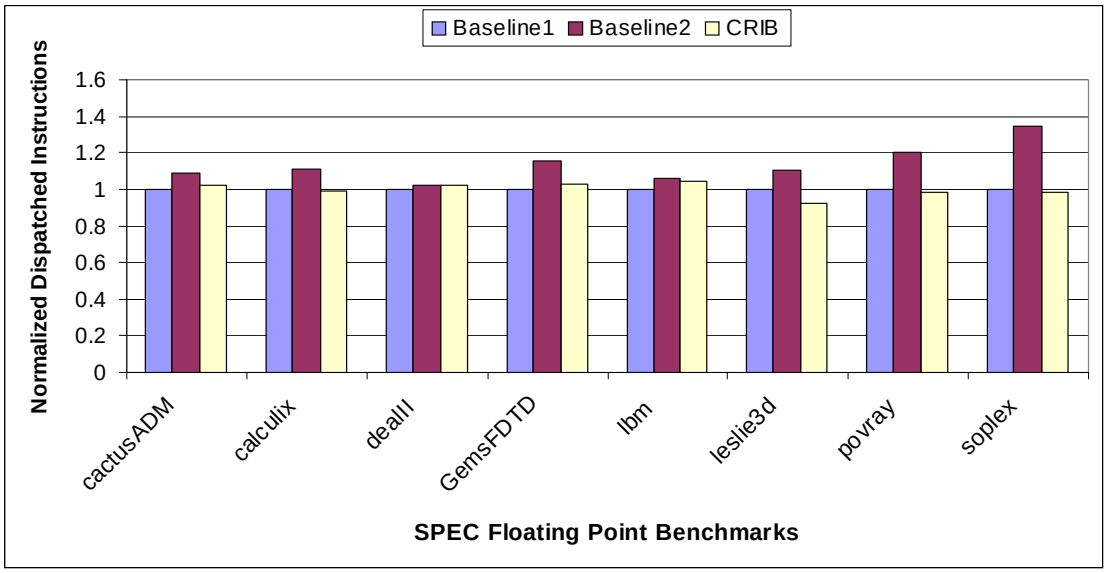


FIGURE 6-5. Dispatched Instructions.

to be redispatched on a pipeline flush. A larger instruction window can also introduce more flushes as more branch predictions have to made, as well as an increased probability for load-store mis-ordering. Since CRIB has a similar window size to baseline1, its number of dispatched instructions is also similar.

Figure 6-6 shows the total number of pipeline flushes per ten thousand instructions, categorized into branch misprediction flushes, load-store mis-ordering flushes, and

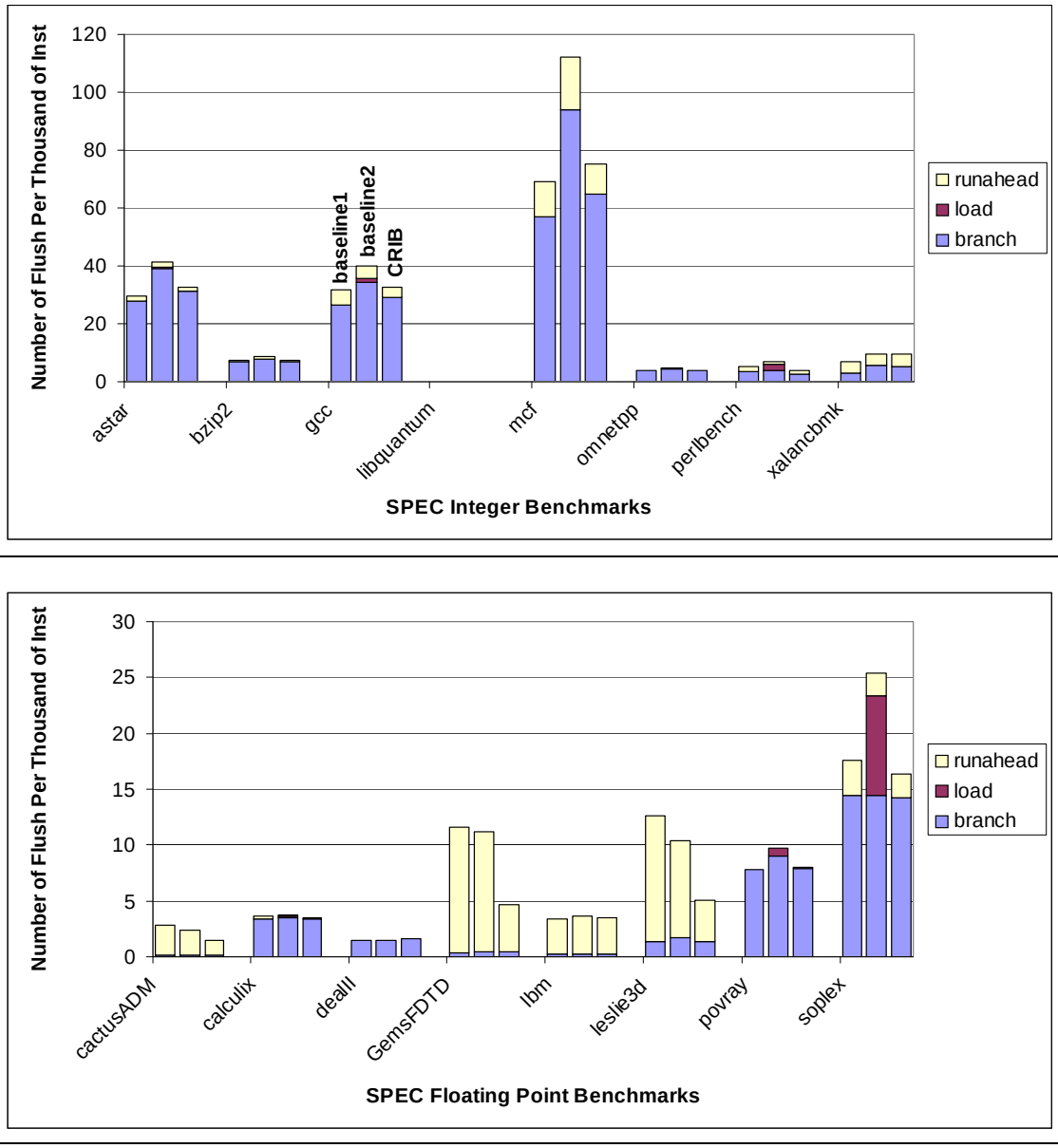
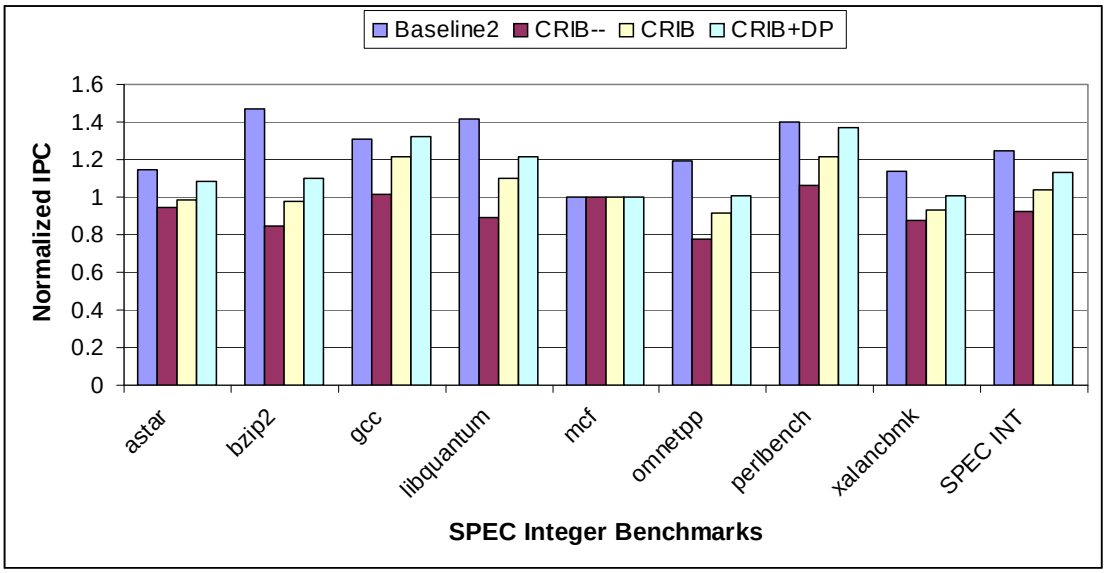


FIGURE 6-6. Total Pipeline Flushes.

runahead flushes. As expected, the increase in the number of dispatched instructions correlate well with the number of pipeline flushes. Mcf, which has the biggest increase in dispatched instructions, also has a large number of pipeline flushes, while libquantum, which has no increase in dispatched instructions, also does not have any flushes. The number of branch mispredictions or load flushes are very high for some benchmarks, especially for mcf. This is caused by the high number of runahead episodes for that particular bench-



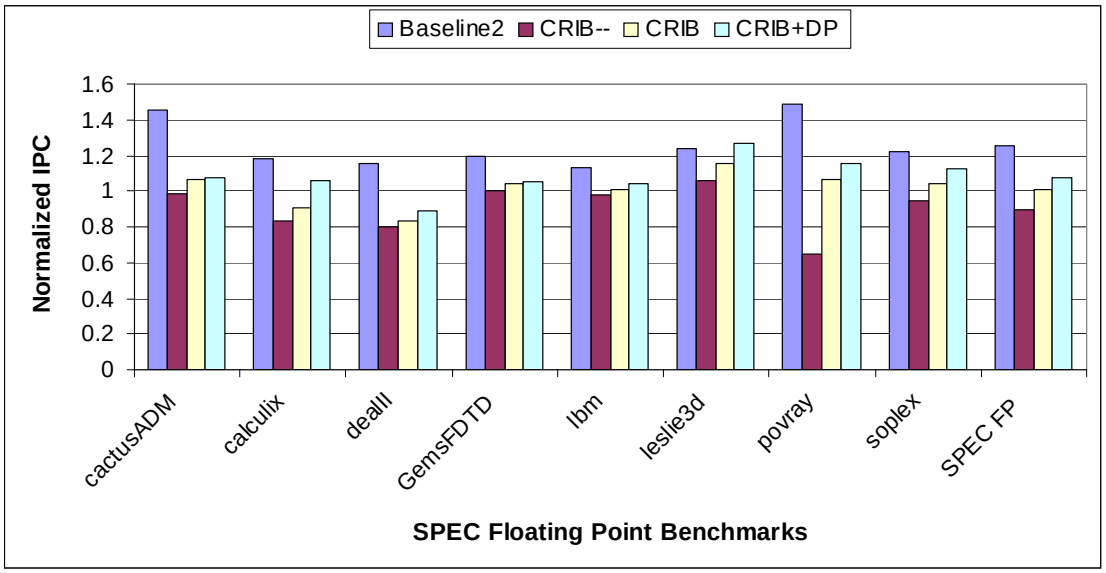


FIGURE 6-7. Normalized IPC Comparison.

mark. The higher the number of runahead episodes, the more branch misprediction and load flushes that can occur.

6.2.2 Performance Comparison

Figure 6-7 shows the IPC comparison between CRIB and baseline2 machine. IPC is normalized into baseline1. For CRIB, we show the IPC for CRIB--, CRIB, and CRIB +

DP. CRIB -- represents CRIB without the advantage of a shorter pipeline and replay recovery for load-store disambiguation. Thus CRIB-- is largely similar to baseline1 but with weaker dependent scheduling, i.e. CRIB can only wake-up and execute dependent instructions within a partition in back-to-back cycles, while baseline1 can wake-up and schedule any dependent instructions in the reservation station in back-to-back cycles. On average the IPC of CRIB-- is 9% less than baseline1 due to this limitation. There are some benchmarks in which the IPC is higher than baseline1 due to more integer ALU resources.

As the execute pipeline is shortened and the aggressive load-store disambiguation with replay recovery is used, the IPC increases exceeding baseline1's. The IPC improvement of CRIB compared to CRIB-- is roughly 12%. As deep pipeline (DP) is added, dependency wake-up becomes more efficient and the IPC increases further, approaching baseline2 IPC. With deep pipeline, CRIB's IPC is 12% less than baseline2 IPC on average.

In general, CRIB's IPC is slightly lower than baseline2 because CRIB has a smaller window and the fact that CRIB requires all instructions in a partition to finish before retiring the partition. Further reducing the reach of the already-small instruction window. CRIB's IPC suffers for those benchmarks that have an instruction causing a CRIB partition to not retire for a while, such as load instruction, floating-point instructions, or, even worse, load misses. However, some of the benchmarks, such as gcc, perlbench, and leslie3d, show comparable or better performance compared to baseline2. The reason is that those benchmarks have a false positive load-store reordering prediction in the main loop that cause the load instruction to stall until the predicted aliasing store from the previous loop iteration retires. In addition to that, leslie3d has a high percentage of par-

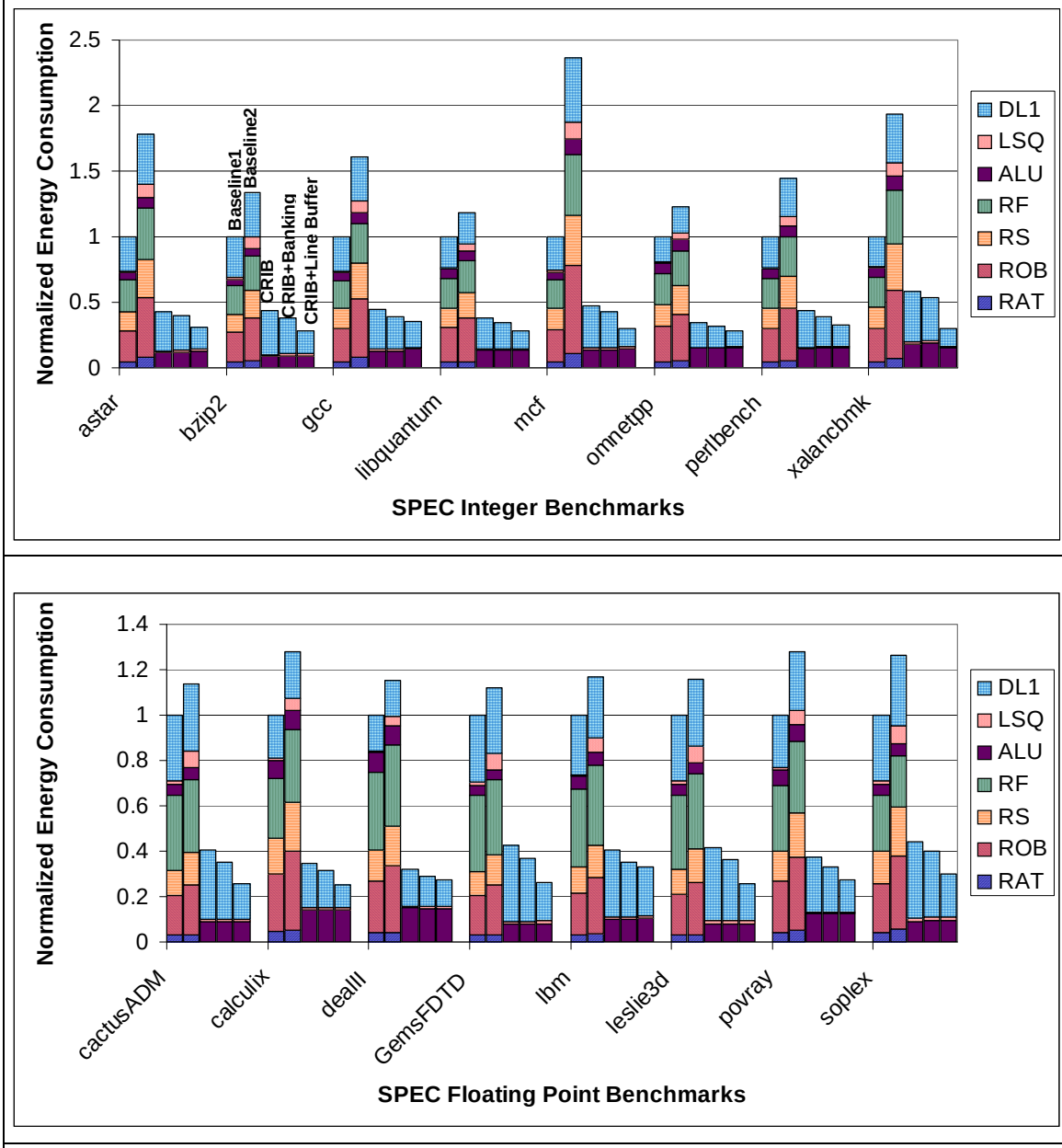


FIGURE 6-8. Energy Comparison for Cache Optimization.

tial load instructions that have to be serialized. By removing the register renaming, CRIB does not require those partial load instructions to be serialized, thus providing more instruction level parallelism compared to a conventional out-of-order processor. A more detailed performance analysis for each of the benchmark is presented in Section 6.4.

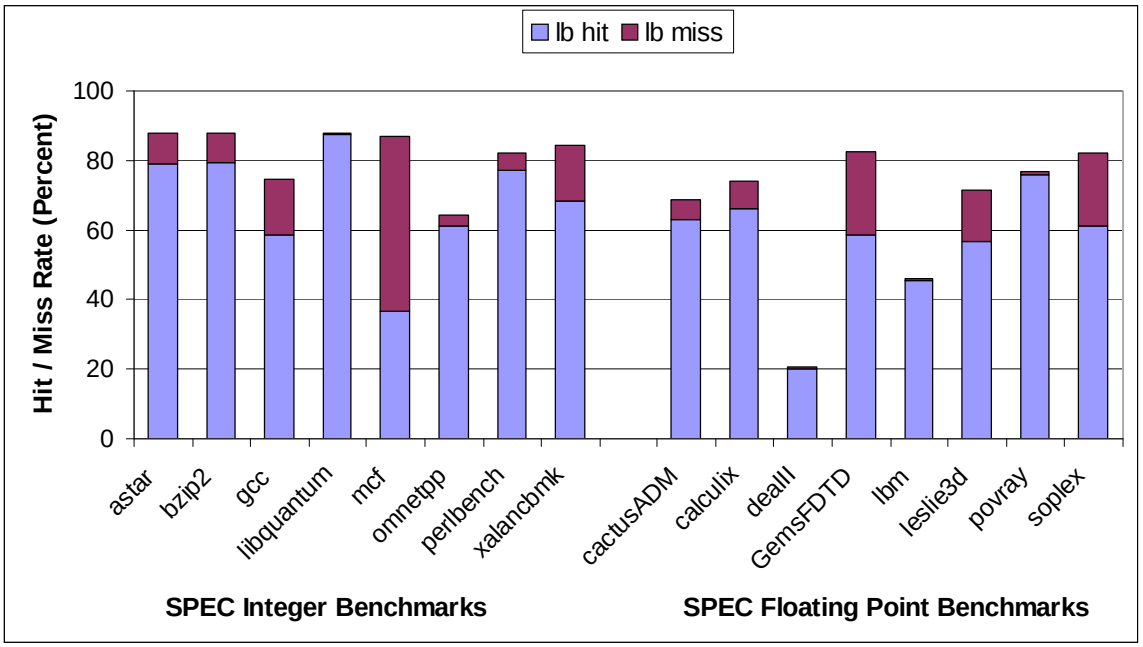


FIGURE 6-9. Hit Rate of Line Buffers.

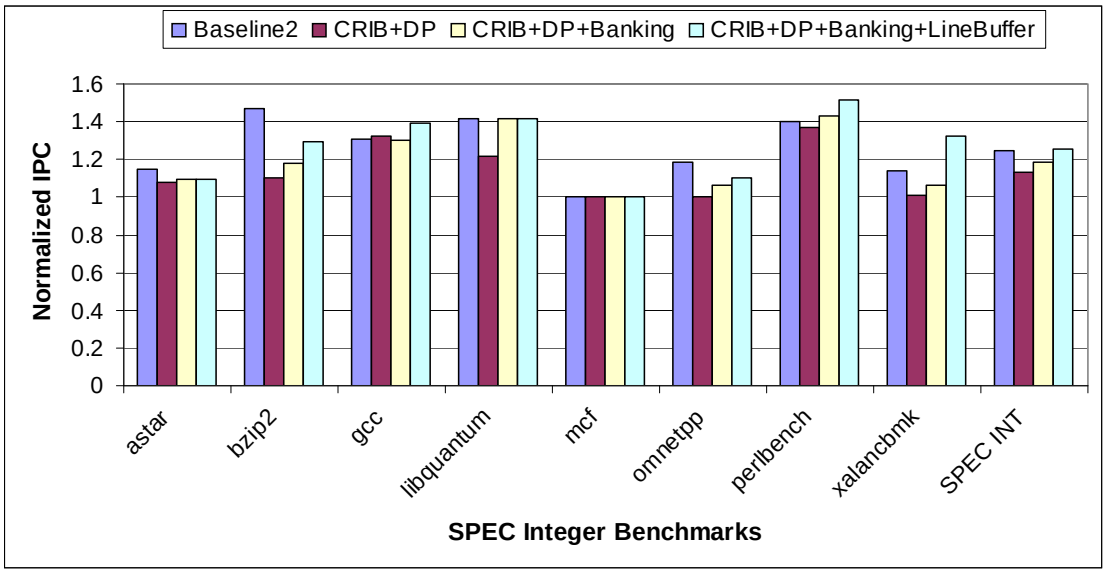
6.3 Cache Optimization

In this section, we analyze the energy saving improvement that is provided by the two cache optimizations being used, cache banking and line buffers.

6.3.1 Energy Saving

Figure 6-8 shows further energy saving that is obtained by employing the two cache optimization techniques. In the figure, there are five bars shown for each benchmark, those bars are for baseline1, baseline2, CRIB, CRIB with cache banking, and CRIB with cache banking as well as line buffers.

The addition of cache banking further reduces CRIB total energy by 3%-4% compared to baseline1. This energy reduction mainly comes from the fact that a 4-banked cache has slightly smaller energy per access compared to 1-banked cache. An additional



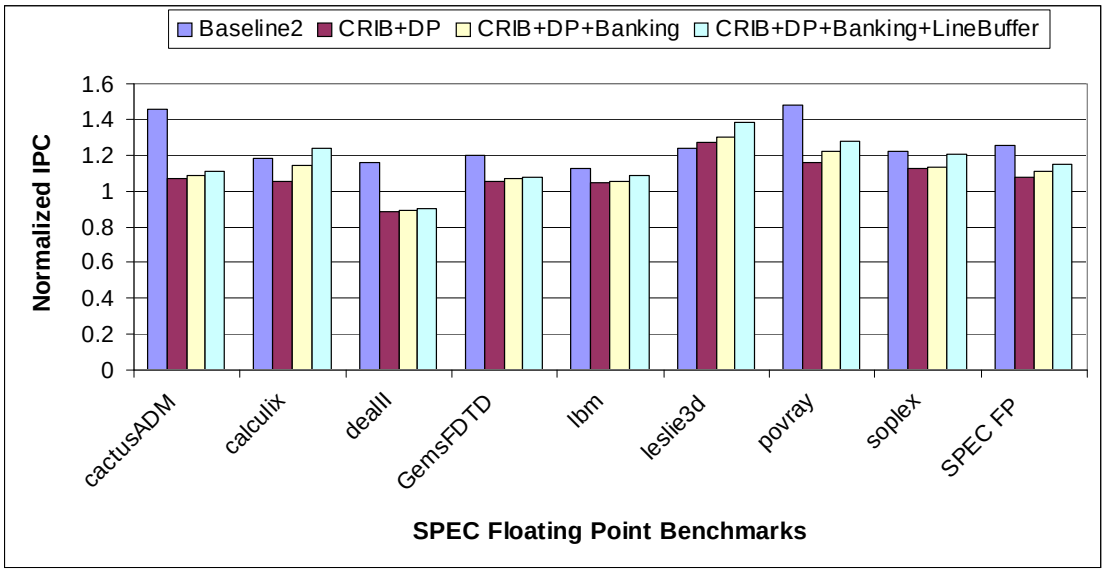


FIGURE 6-10. IPC Comparison for Cache Optimization.

4%-7% of energy savings, depending on the benchmark, is gained by employing the line buffer technique. With the additional energy saving from the cache optimization, CRIB's energy consumption is around 20% - 23% of baseline2's energy consumption.

Figure 6-9 shows the line buffer hit rate. The hit rate is further categorized into 'lb hit' where the set to access is in the line buffer and the line buffer results in cache hit. The other category is 'lb miss' where the set to access is in the line buffer, but the tag check

results in a cache miss. In both cases, the subarray is not accessed. As seen, with a total of 16 line buffers, one for each subarray, the hit rate is quite high, close to 90% of all accesses. A higher hit rate results in higher energy saving. Benchmarks such as dealII and lbm that do not have a high line buffer hit rate do not have as high of additional energy saving from the line buffer.

6.3.2 Performance Effect

Figure 6-10 shows IPC comparison after cache optimization is added to CRIB. The additional bandwidth provided by cache banking increases CRIB's IPC by 5% for SPEC integer and 2% for SPEC floating point on average. The addition of line buffers gives another 7% and 5% of IPC increase for SPEC integer and SPEC floating point respectively. With the cache optimizations, CRIB's average IPC for SPEC integer is comparable to Baseline2's average IPC. For SPEC floating point, CRIB's average IPC is still 8% less than Baseline2's average IPC. Overall, cache optimization techniques help SPEC integer more than SPEC floating point. The reason is that most of the long latency instructions in SPEC integer are load instructions, thus cache optimization techniques are able to reduce the latency of the load instructions. In contrast, most of the long latency instructions in SPEC floating point are floating point operations, which latency cannot be reduced by the cache optimization techniques. There are benchmarks that suffer significant IPC drops even after cache optimizations are applied, such as bzip2, omnetpp, cactusADM, dealII, GemsFDTD, and povray. There are also some benchmarks whose IPC are significantly higher than baseline2 such as perlbench, xalancbmk, and leslie3d. Those benchmarks will be analyzed in detail in the next section.

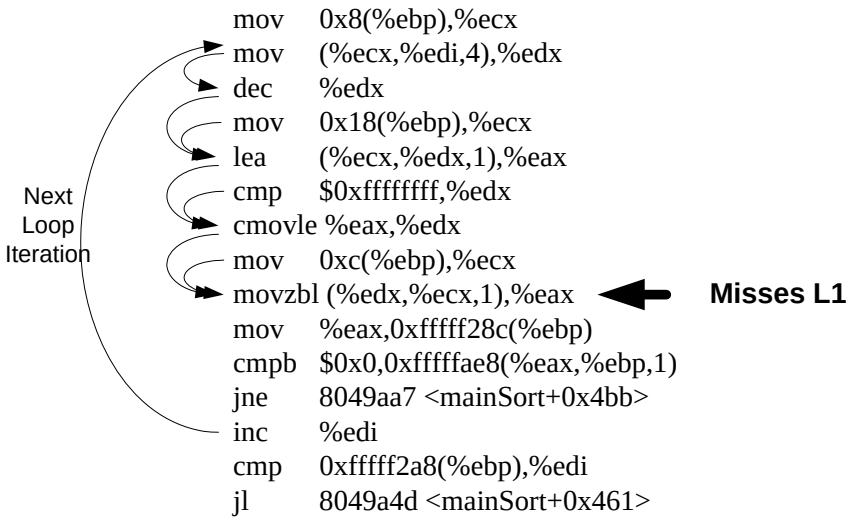


FIGURE 6-11. Bzip2 Loop.

6.4 Detail Benchmark Analysis

In this section we try to analyze benchmarks whose results are much different from the average. In particular, we explain why the performance of such benchmarks running on CRIB is significantly more or significantly less than the baseline machines. Among those benchmarks are bzip2, libquantum, mcf, omnetpp, perlbench, cactusADM, dealII, GemsFDTD, leslie3d, and povray.

6.4.1 Bzip2

Bzip2 IPC starts with 0.96 for baseline1. Increasing the window for baseline2 increases the IPC by 45% into 1.41. The IPC of CRIB+DP is 1.06, 5% higher than baseline1's. After the cache optimizations are applied, Bzip2's IPC is 1.24, still less than baseline2 IPC. A quick look at the window size sensitivity study in Figure 6-1 shows that Bzip2's IPC keep increasing significantly until 64 CRIB entries, at which point the IPC is slightly above Baseline2's IPC.

Figure 6-11 shows the loop in Bzip2 program that is executed repeatedly during execution. The loop consists of 15 instructions. Dependency arcs are shown in the figure. One of the load instructions in the loop repeatedly misses the L1 cache and hits in the L2 cache. Each loop iteration is independent of each other except for the EDI register that is used to address a load instruction in the beginning of each iteration. However, since the EDI increment is not a bottleneck, one can say that each iteration is virtually independent of each other. There we can see that a larger window would benefit the execution of this loop considerably, especially with the missed load in the middle of the loop. With a larger window, the latency of the missed load can be hidden by pipelining the loop execution. These factors explain why CRIB does not perform as well as baseline2.

6.4.2 Libquantum

Libquantum is a high IPC benchmarks. Its IPC on baseline1 is 2.12. It increases by 42% on baseline2 to 3 IPC. CRIB+DP has 2.57 IPC, 20% less than baseline2. However, as cache banking is applied, CRIB's IPC increases to 3, the same as baseline2's IPC. Sensitivity study shows that the IPC for 16 entries is still slightly less than the peak IPC, which occurs starting from 32 entries. Thus the problem here is the window size.

Figure 6-12 shows the loop in libquantum that is called repeatedly for the 100M instructions being executed in the timing simulator. It is a very short loop with nine instructions in it. The figure is annotated with 'Dispatch', 'Execute', and 'Commit' cycles for each of the instruction. As shown, after 16 cycles, the dispatch cycles for CRIB slips by 2 cycles, causing the execution to also get delayed. Because there is a dependency between a load and an integer operation in one partition, a CRIB partition containing the particular instructions takes six cycles to finish, causing the next dispatch group into the

			Baseline2 D,E,C	CRIB+DP D,E,C	
1.	mov	0x10(%ebp),%ebx	1,3,6	1,2,7	
2.	movl	$0x0,(%ebx,%eax,4)	1,6,8	1,5,7	
3.	mov	0x8(%ebp),%ebx	1,4,8	1,3,7	
4.	mov	%ebx,%ecx	1,7,8	1,6,7	
5.	mov	$0x1,%edx	2,4,8	2,3,9	
6.	shl	%cl,%edx	2,8,9	2,7,9	
7.	add	$0x1,%eax	2,4,9	2,3,9	
8.	cmp	%edx,%eax	2,9,10	2,8,9	
9.	jl	804a719 <quantum_gate1+0x39>	3,10,11	3,9,10	
10.	mov	0x10(%ebp),%ebx	3,5,11	3,4,10	
11.	movl	$0x0,(%ebx,%eax,4)	3,8,11	3,7,10	
12.	mov	0x8(%ebp),%ebx	3,6,11	3,5,10	
13.	mov	%ebx,%ecx	4,7,12	4,6,11	
14.	mov	$0x1,%edx	4,6,12	4,5,11	
15.	shl	%cl,%edx	4,8,12	4,7,11	**Dispatch Slips by 2 Cycles**
16.	add	$0x1,%eax	4,6,12	4,5,11	
17.	cmp	%edx,%eax	5,9,13	7,8,12	
18.	jl	804a719 <quantum_gate1+0x39>	5,10,13	7,9,12	
19.	mov	0x10(%ebp),%ebx	5,7,13	7,8,12	
20.	movl	$0x0,(%ebx,%eax,4)	5,10,13	7,11,12	
21.	mov	0x8(%ebp),%ebx	6,8,14	9,10,15	
22.	mov	%ebx,%ecx	6,9,14	9,13,15	
23.	mov	$0x1,%edx	6,8,14	9,10,15	
24.	shl	%cl,%edx	6,10,14	9,14,15	

FIGURE 6-12. Libquantum Loop.

particular CRIB partition to get delayed by two cycles. With the addition of cache banking, the bandwidth is doubled and the latency is reduced. Without cache banking, a cache access effectively takes one cycle for address generation and two cycles for the cache access. With cache banking, a cache access only takes 0.5 and 1.5 cycles for address generation and the cache access, effectively reducing the load-to-use latency. With this optimization, the same CRIB partition that previously takes six cycles will now takes four cycles, eliminating the dispatch stalls for the next CRIB.

6.4.3 Mcf

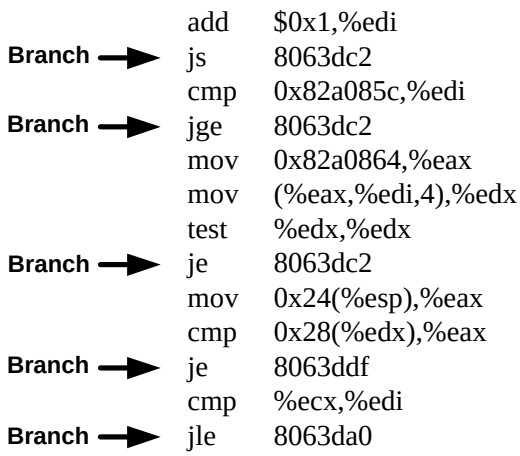


FIGURE 6-13. Omnetpp Loop.

Mcf is a benchmark with very high miss rates going to memory. Its IPC on baseline1 is 0.222. Its IPC does not increase nor decrease as the program is run on different machines, such as baseline2 and CRIB. Because mcf has a high miss rate, most of its time is spent waiting for the miss to be back from the memory. Without runahead, mcf will benefit from larger windows due to pipelining of the load misses. Since we use runahead in our study, baseline1, baseline2, and CRIB have a similar runahead window size to discover further misses. Because of this, mcf does not show any performance improvement going from baseline1 to baseline2 that has a much larger window. It also does not show any improvement or degradation on CRIB.

6.4.4 Omnetpp

Omnetpp has an IPC of 1.68 on baseline1. Its IPC increases to 1.99 for baseline2 and to 1.69 for CRIB+DP. The addition of cache optimizations increases the IPC further to 1.85, still 7% slower than baseline2. A closer look at the occupancy graph in Figure 6-3 shows that CRIB's occupancy of three or more instructions only happens 55% of the time.

Figure 6-13 shows one of the main loops that is repeated many times over in the

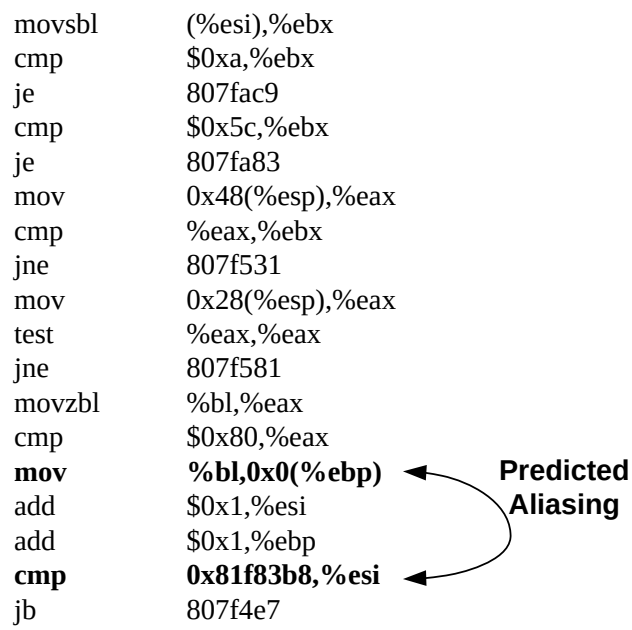


FIGURE 6-14. Perlbench Loop.

detailed timing simulation. As shown, while only having thirteen instructions, this loop has five branches. CRIB can only have one branch in each partition. Thus, these instructions have to be spread into five partitions, further exacerbating CRIB's problem of having a small window. However, it can be solved by allowing more than one branch in a CRIB partition. Only mispredicted branches and branches with non-saturating prediction need to update the branch predictor and those are the rare cases. To accommodate those rare cases, a buffer for branch predictor update can be added in the front-end part of the machine.

6.4.5 Perlbench

Perlbench has an IPC of 0.80 on Baseline1. Baseline2 increases the IPC by 40% to 1.13. CRIB+DP increases the IPC by 37% to 1.10, similar with baseline2's. Cache optimization further increases the IPC to 1.21, slightly higher than baseline2's IPC.

While CRIB has disadvantage of a small window, it has the advantage of employ-

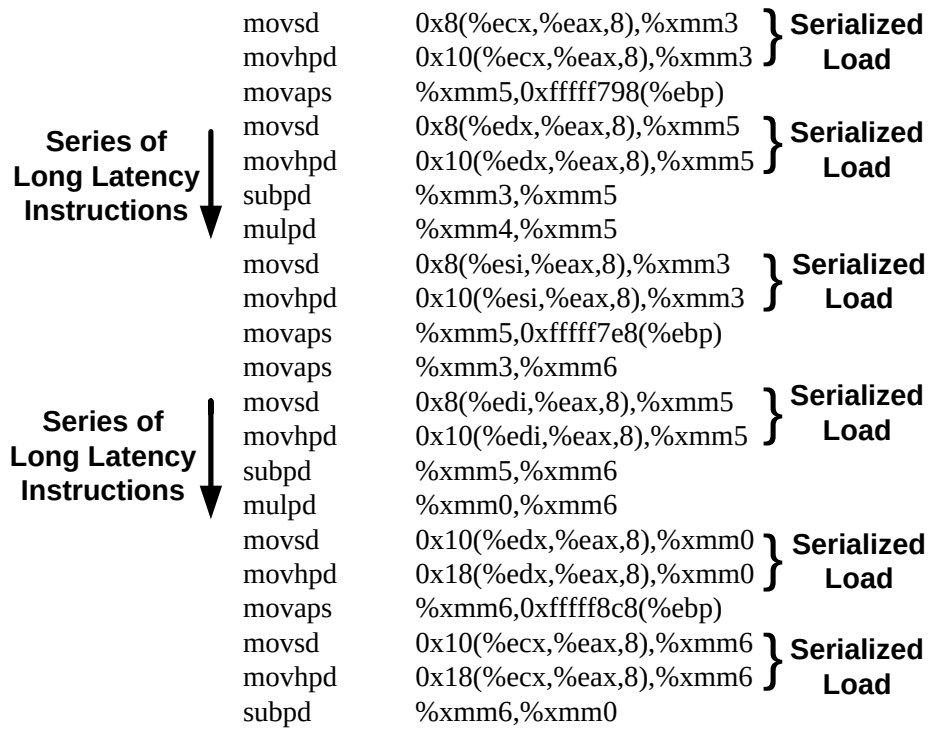


FIGURE 6-15. cactusADM snippet.

ing aggressive load-store reordering. In this particular benchmark, the advantage of aggressive load-store reordering is able compensate for its small window. Figure 6-14 shows the loop snippet of perlbench. The predicted aliased load and store is annotated in the figure. In the baseline2 model, this false positive prediction delays the issue of the load until the store instruction retires from the reorder buffer, thus resulting in a long wait before the load can issue. For this case, a better predictor or the ability to unlearn of prior prediction might be helpful to avoid repetitive false positive predictions in future prediction.

6.4.6 cactusADM

Baseline1's IPC for cactusADM is 0.58. It increases by 45% with baseline2, while CRIB+DP increases the IPC only by 7%, much lower than the IPC increase by baseline2.

The IPC further increases by 11% after cache optimization techniques are applied, still much lower than baseline2 IPC. The main reason of the slowdown compared to baseline2 is CRIB's window size. Figure 6-15 shows the code snippet of cactusADM. The code snippet is interesting that it contains program characteristics that give advantage and disadvantage to CRIB architecture. In the Figure 6-15, it can be seen that the snippet has a high percentage of serialized loads. These serialized load instructions happen because XMM registers are 128 bits, wider than the load instruction result that is 64 bits. Each load instruction only fills a portion of the XMM register, the low portion or the high portion. Because of explicit register renaming, each of the load instructions allocates physical register for the same logical xmm register. Thus, the second load instruction has to wait for the first one to complete and combine the result with its own result and write it to its allocated physical register. On the other hand, CRIB does not explicitly rename the registers. Hence, the second load instruction does not have to wait for the first one to be done since the location of the register does not change. Because of this, CRIB has the advantage when the code has many of the serialized loads.

However, the code shown in Figure 6-15 also contains a series of long latency instructions that slows the turn around rate in CRIB, further reducing the instruction level parallelism of CRIB's small window. With a large window, baseline2 is able to pipeline the long latency instructions as well as the many serialized loads. Thus providing a significant speedup over baseline1.

6.4.7 dealII

DealII's IPC for baseline1 is 0.93. With baseline2, it increases by 16% to 1.08. The IPC for CRIB+DP is 0.83, less than baseline1's IPC. With cache optimizations, the IPC

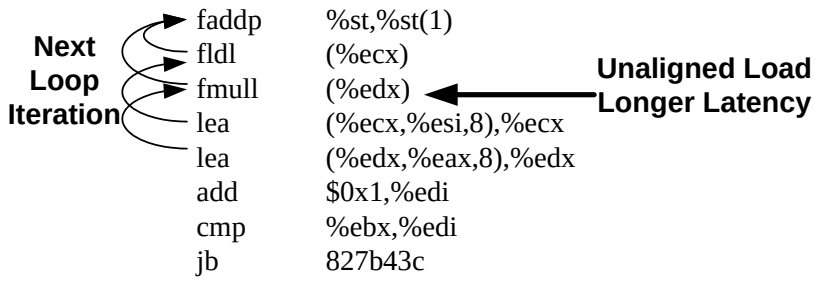


FIGURE 6-16. dealII Loop.

increases slightly to 0.84, still less than baseline1's IPC.

Figure 6-16 shows the loop executed during simulation. DealII is compiled using x87 rather than SSE because SSE compilation of dealII does not run correctly on our simulator. As shown, the loop is fairly short with only eight instructions. The loop is also pipelined so that the result of one iteration will be used for the next iteration. With these instructions, the reason for the low IPC is the small window and the fact that new instructions cannot be dispatched until all instructions in CRIB are finished. It is made worse by the fact that one of the load instruction is unaligned and requires additional latency, and is followed by a floating-point computation. The load instruction followed by floating-point computation reduces the turn-around rate of CRIB partitions significantly.

6.4.8 GemsFDTD

GemsFDTD is similar to cactusADM. Baseline1 has an IPC of 0.37. Baseline2 only gives an IPC increase of 20% while CRIB+DP increases the IPC by 5%. Cache optimization further increases CRIB's IPC by 8%. A closer look at GemsFDTD code snippet shown in Figure 6-17 explains the behavior. Similar to cactusADM, GemsFDTD also has high percentage of long latency instruction groups that reduce the instructions turn around rate in CRIB. While GemsFDTD also has many serialized load instructions, the advantage

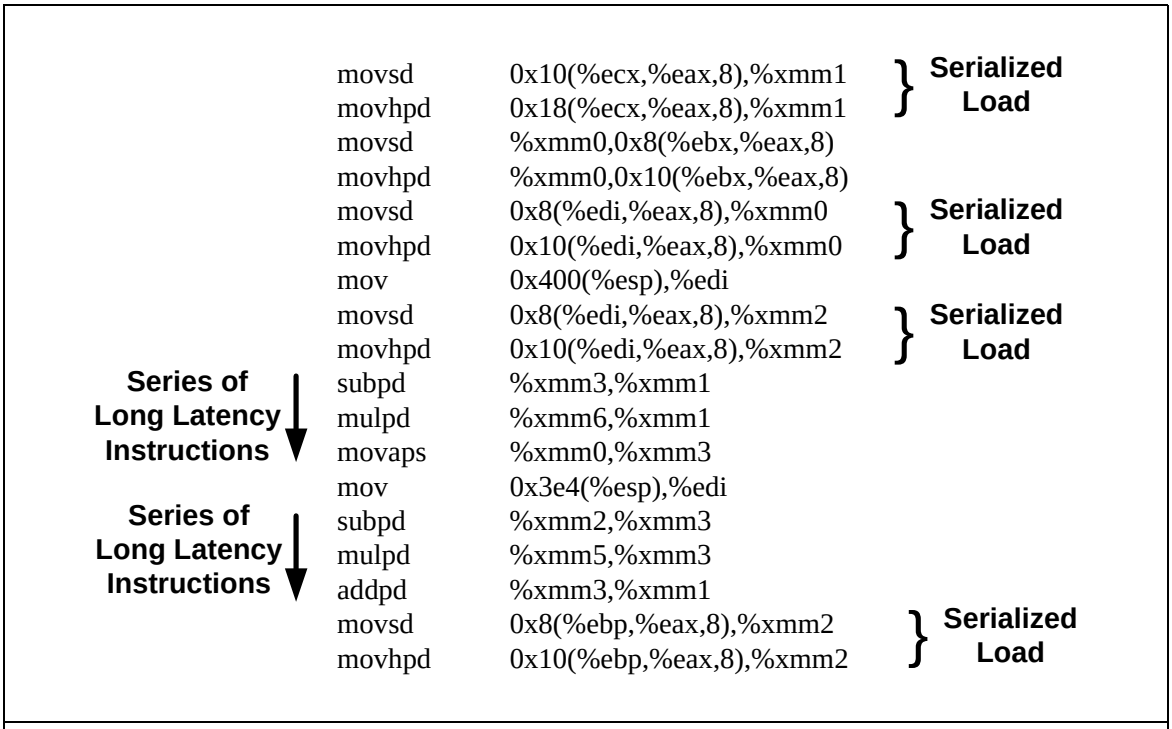


FIGURE 6-17. GemsFDTD Code Snippet.

that CRIB has for those load with partial write cannot overcome the disadvantage of its small window.

6.4.9 leslie3d

Leslie3d is another benchmark that has many serialized load instructions. In addition to that there is a false positive aliasing prediction in one of the main loop as shown in Figure 6-18. Figure 6-18 shows the main leslie3d loop that is called repeatedly in the detailed timing simulation. The predicted aliasing disables the load instruction to be issued until the predicted aliasing store instruction from the previous loop iteration is retired. Thus reducing the opportunity to pipeline the dynamic instances of the loop. Leslie3d has baseline1 IPC of 0.362. Baseline2 increases the IPC by 24% while CRIB+DP gives an IPC boost of 27%. With cache optimization techniques, the IPC of CRIB increases into 38% over baseline1.

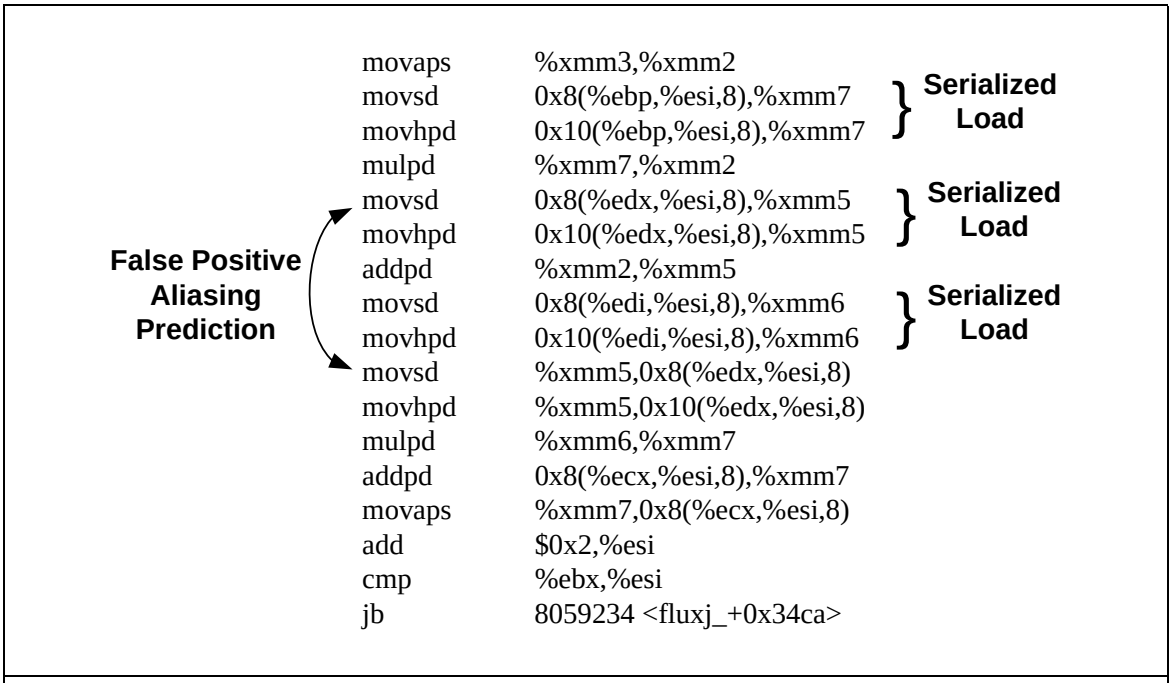


FIGURE 6-18. Leslie3d Loop.

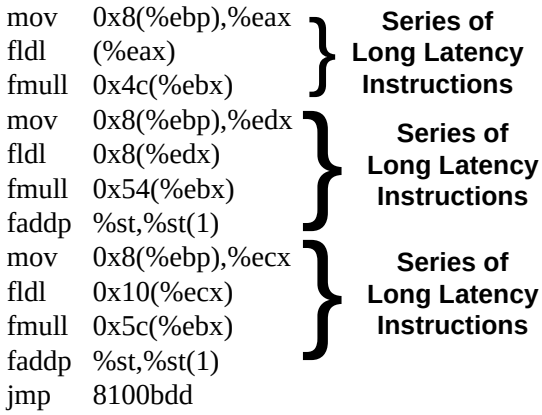


FIGURE 6-19. povray snippet.

6.4.10 povray

Povray has an IPC of 0.912 for baseline1. It increases to 1.353 for baseline2. For CRIB+DP, the IPC is 1.051. After cache optimizations, CRIB's IPC increases into 1.163, still less than baseline2's IPC. Figure 6-19 shows a code snippet from povray's loop that is executed during detailed simulation. From the code snippet, it can be seen that the instruc-

Table 6-1: Program Characteristic Impact on CRIB.

Program Characteristic	+ / -
Long latency instruction: cache misses, unaligned loads	-
A chain of back-to-back dependent instructions	-
Hard to predict load-store aliasing (resulting in false positive prediction)	+
Partial register writes	+

tions forms a series of long latency instructions started by a load instruction. In the snippet, one of the instructions streams has three load instructions in a row followed by two floating point operations, and each of them is the consumer of the previous instruction. One would notice that if four of these instructions occupy a CRIB partition, the partition will have to execute for along time, reducing the turn around rate for the insertion of new instructions significantly.

6.4.11 Benchmarks Analysis Summary

In general, there are common program characteristics that cause CRIB to experience slow-down or speed-up compared to conventional out-of-order machines. Performance degradation usually occurs when there are long latency instructions causing a CRIB partition to be occupied for a while. This kind of situation exacerbates the problem of CRIB having small window. Whenever the problematic long latency instructions are load instructions, cache optimization techniques employed can help reduce the problem. However, long latency floating-point instructions are harder to handle.

On some benchmarks, CRIB shows performance improvement over the conventional out-of-order machine. It usually happens when 1) the load-store disambiguation predictor keeps giving a false positive prediction for a particular load instruction, and 2) when the executed programs have a high percentage of partial register writes. The impact of various program characteristic to CRIB's performance are summarized in Table 6-1

6.5 Chapter Summary

In this chapter we present a detailed performance and energy evaluation of our CRIB implementation. We also show the performance and energy evaluation for CRIB with cache optimization techniques. Our sensitivity studies show that the chosen CRIB configuration is an optimal configuration for most benchmarks. We show that the average energy saving in the execution core for SPEC CPU 2006 integer and floating-point are 72% and 67% respectively compared to baseline2. Adding cache optimizations further increases the energy saving to 80% and 77% respectively. CRIB's IPC is competitive compared to baseline2. Without cache optimizations, the IPC is 10% and 14% lower than Baseline2's for SPEC integer and SPEC floating point respectively. With cache optimizations, the IPC for SPEC integer is 1% higher than Baseline2's. For SPEC floating point, the IPC is 8% lower than Baseline2's IPC after cache optimization techniques. In this chapter we also explains in detail why CRIB performs better or worse than conventional out-of-order machine for certain benchmarks.

Chapter 7

Conclusion

Traditionally, power consumption in high-performance CMOS processors was only a secondary design constraint because supply voltage scaling across technology generation had a quadratic effect in reducing chip power consumption. Unfortunately, recent and future nanometer CMOS technology processes no longer provide similar voltage scaling, leading to a serious power bottleneck for aggressive deeply-pipelined processors. The power constraint problem is exacerbate by the industry trend toward many-core processors. While technology scaling allows the number of cores in a chip to keep increasing, chip power budget does not increase as fast; this further limits the power budget available for each core.

Prior work has proposed reducing power consumption in different parts of the cores. However, most of the work focuses only on specific structures. While successful in reducing the power for the particular structure, the power reduction is not significant at the core level. A closer look at chip power distribution shows that a large percentage of chip power is spent on operand delivery. Power consumption for operand delivery is even larger than the power needed to do the execution itself. We try to reconstruct the way out-of-order execution is done in order to reduce the power consumption associated with operand delivery.

This thesis presented *CRIB: Consolidated Rename, Issue, and Bypass,* an approach to do out-of-order execution without explicit register renaming. By removing explicit register renaming, several supporting structures needed for operand delivery can be elimi-

nated. Hence, power consumption related to operand delivery can be dramatically reduced. The CRIB design also allows lightweight recovery for mis-ordered load instructions. This recovery allows CRIB to use aggressive load-store reodering rather than using prediction. Due to the positional structure of CRIB, branch misprediction recovery is also simpler than conventional out-of-order machines. Lastly, CRIB removes separation of data and dependency linking used in conventional out-of-order machines, resulting in the removal of speculative scheduling. With the removal of speculative scheduling, various cache optimizations that were not attractive for conventional machines, due to the added latency non-determinisn, can be employed.

The first part of this thesis describes the concept of CRIB and its implementation detail. It also describes cache optimizations technique, cache banking and line buffers, used in our CRIB design. Our detailed energy evaluation indicates that the average energy saving in the execution core for SPEC CPU 2006 integer and floating-point are 72% and 67% respectively compared to a conventional out-of-order machine with a large instruction window. Adding cache optimizations further increases the energy saving to 80% and 77% respectively. Assuming that the front-end, clock-tree, and L2 cache consume 50% of total core power and that the energy consumption for them remains constant, the energy saving translates into 40% and 38% of core energy consumption. Without cache optimizations in place, CRIB's IPC is 10% and 14% lower than conventional out-of-order machine with large window for SPEC integer and SPEC floating point respectively. With cache optimizations, the IPC is 1% higher and 8% lower than a conventional out-of-order machine for SPEC integer and SPEC floating point respectively. CRIB also has smaller

comparable area for the affected structures, 2.188 mm^2 compared to 2.507 mm^2 for a conventional out-of-order machine.

For some benchmarks such as bzip2, cactusADM, dealII, GemsFDTD, and povray, CRIB performance is not as high as a conventional out-of-order machine. The reason for performance reduction are the small window of CRIB and the slow turn-around rate. Due to area limitations, we only use 16 entries. Because CRIB does not allow a partition to retire until all instructions are finished, long latency instructions such as load instructions and floating-point instructions cause the slow turn-around rate. The slow turn-around rate caused by load instructions can be improved by cache banking and line buffers. Thus, we obtain IPC improvement with the cache optimization techniques for the problematic integer benchmarks.

There are also some benchmarks where CRIB performs well, even better than conventional machine. This occurs for two reasons. The first is the fact that CRIB uses aggressive load-store reordering rather than relying on a predictor. For some benchmarks, this predictor results in a high percentage of false positive predictions, causing load instructions to get delayed even when there is no aliasing. Since CRIB does not use a predictor, it does not suffer from the same problem as a conventional machine. The second reason is when there are a lot of partial load instructions in the instruction stream. Since register renaming moves the physical location of a register around, a load instruction that only changes a portion of a register while leaving the other portion intact has to create a dependency with the destination register. In SSE instructions, register width is 128 bits while load instruction width is only 64 bits. Thus SSE applications have many partial load instructions that have to be serialized in their execution.

In summary, this thesis shows that by reconstructing conventional out-of-order machine, we can save significant energy while maintaining competitive performance and area requirements.

7.1 Future Work

There are several areas of future work that can be explored. We highlight several of them here. Our result shows that the small window of CRIB causes performance degradation for some of the benchmarks. We discuss possible solution in Section 7.1.1. While CRIB reduces execution core energy significantly, it does not change the front-end part of the processor. The fact that CRIB does not need explicit register renaming can be used to extend loop stream buffer to save front-end energy as discussed in Section 7.1.2.

7.1.1 CRIB's Window Size

The fact that we try to match the area of a conventional design limits the number of entries in our CRIB design. Our result shows that the small window causes significant performance degradation to some of the benchmarks even after the application of cache optimization techniques. This degradation mainly occurs for floating-point benchmarks that have a large number of long latency instructions.

In the current design, CRIB has an integer partition and a floating-point partition. Most integer instructions occupy only the integer partition leaving the floating-point entry empty. Similarly, most floating-point instructions occupy only the floating-point partition leaving the integer counterpart empty. Only floating-point or SSE load and move instructions occupy both partitions. This design is quite wasteful in terms of space. A simple rearrangement could potentially increases the window size.

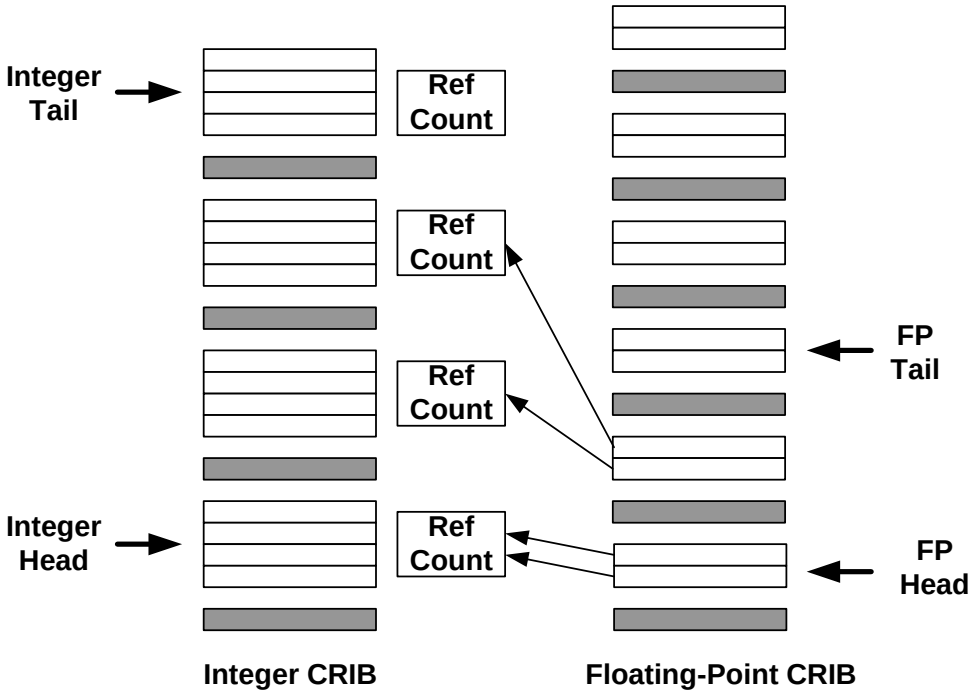


FIGURE 7-1. Disjoint CRIB Diagram.

Figure 7-1 illustrated the potential solution that can be explored. Integer CRIB and floating-point CRIB can be designed as disjoint structures. An entry in integer CRIB no longer has a corresponding entry in floating-point CRIB. In this disjoint CRIB, integer CRIB does not waste an empty entry in the floating-point partition and vice versa. The number of partitions does not have to be the same between integer and floating-point CRIB. The head and tail pointer also do not have to be the same. Integer and floating-point partitions can be resized according to their needs; Figure 7-1 illustrates that the integer partition has four entries while the floating-point partition only has two entries. To maintain precise exceptions, reference counter can be added to the integer CRIB to make sure that it is not reclaimed before the floating-point counterparts are finished. With the disjoint CRIB, a 16-entry integer CRIB and a 16-entry FP CRIB could potentially host 32 instructions, doubling the original capacity of 16 instructions.

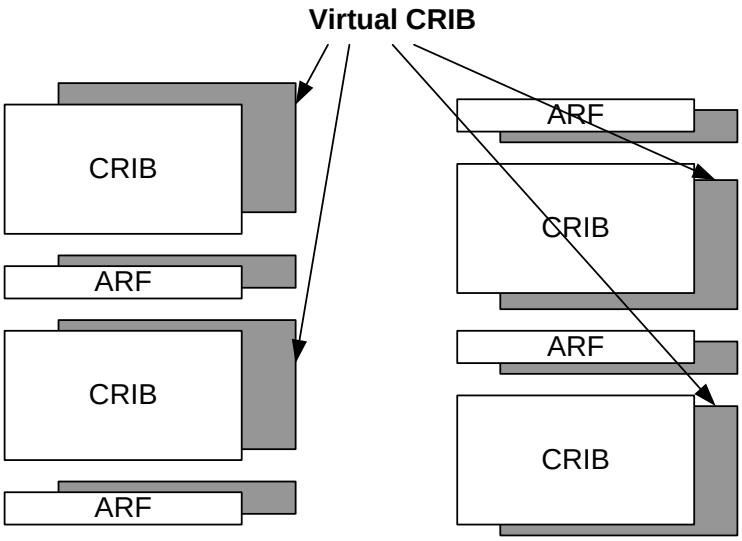


FIGURE 7-2. Virtual CRIB Diagram.

Another potential solution that can be explored to increase CRIB's window size is by having a virtual CRIB as shown in Figure 7-2. In the current design, CRIB partitions already have shadow copies of the ARFs to be used during runahead. The two copies of the ARF can be exploited further to double the window size during long latency instructions, i.e. load instructions that miss the L1 but hit in the L2. This support can be expanded to cover floating-point instructions also. In order to achieve this, we need to add the virtual CRIB, which is basically another copy of CRIB's storage, shown as the shaded region in Figure 7-2. Whenever the CRIB is full and the head of the CRIB has long latency instructions that are not finished, the ARF can latch the result as is. The virtual copy can then be used to dispatch more instructions and the shadow ARF can now be used to propagate the instructions to the next CRIB. Whenever the long latency operation is done, the execution is switched back into the true CRIB, where the ARF now captures the result that was previously incomplete. Once all the true ARFs capture the needed result, the virtual CRIB

can now be the true CRIB while the true CRIB will now serve as virtual CRIB for the next long latency instructions.

7.1.2 Front-end Energy Reduction

A large portion of chip power consumption is the front-end energy. While we have significantly reduce the execution core and data cache power, we do not modify the front-end model. Since CRIB does not need register renaming, prior work on caching the instruction stream, such as loop buffers and branch target caches can be extended and paired with CRIB. The simplest solution would be to have a separate loop buffer and branch target cache that will serve CRIB. A more aggressive solution could potentially merge the loop buffer with the CRIB itself. With the window size expansion in Section 7.1.2 that could potentially quadruple current CRIB's window size, it could be possible to have a static instruction stream in the CRIB and create the dynamic flow as the CRIB traverses the static stream.

7.1.3 Fission and Fusion

SMT and core clustering concepts can also be applied to CRIB. Whenever thread level parallelism is deemed to be necessary CRIB can be split into smaller cores. For example, a four partition CRIB can be split into four cores with one partition each, or two cores with two partitions each. The split need not be balanced , i.e. in the two-core case, one core can have three partitions while the other one can have only one partition. On the other hand, whenever a powerful core is needed, adjacent CRIB can be fused into one core to create a core with a larger window and more resources.

7.1.4 Software Aspect

As explained in Section 6.4, a series of dependent instructions within a CRIB partition results in a slow turn around rate that can potentially hurt the IPC. Thus, splitting dependency chains across CRIB partitions can result in higher turn around rate and increase IPC. On the other hand, splitting the dependency chain too far apart potentially can lengthen the critical path through the dataflow graph and hurt performance. Software such as compiler, virtual machine, or a smart decoder, can create instruction groups that help CRIB hide its small window effect while maintaining dependency distances within reach of single-cycle propagation. A trade-off study could be done to understand the balance between minimizing dependency chains in a partition and the length of dependency distances that are introduced.

7.1.5 Impact on Design Methodology and Flow

The removal of latches between functional units in a CRIB partition omplicates the testing and verification effort. While it is possible to insert four independent instructions to test each of the functional unit inside a partition, it is difficult to isolate whether the fault is in the functional unit itself or in the routing path to the architected register file. One possible solution would be to insert some latches back into the CRIB partition that are only used during testing. The drawback of this solution would be an increase in area. Another solution would be to create a configurable partition that is able to write the results back into some portion of the instruction storage inside each entry, such as the PC portion, during testing. Thus, each functional unit can be tested independently. Minor changes in control signal are needed for this solution.

7.2 Chapter Summary

In this chapter we conclude the dissertation. We re-state the motivation, concept, and implementation of CRIB. We summarize our result analysis for energy, performance, and area. Lastly, we describe several areas of future work that can be explored.

References

[1] ACS Simulation Tools about IVM version 1.0. *http://www.crhc.illinois.edu/ACS/tools/ivm/about.html*

[2] Bochs the Cross Platform IA-32 Emulator. *http://bochs.sourceforge.net*

[3] Standard Performance Evaluation Corporation. *http://www.spec.org/cpu2006*

[4] Anant Agarwal, John Hennesy, and Mark Horowitz. Cache performance of operating systems and multiprogramming. *ACM Transactions on Computer Systems*, 6:393-431, November 1988.

[5] Amit Agarwal, Bipul C. Paul, Saibal Mukhopadhyay, and Kaushik Roy. Process variation in embedded memories: Failure analysis and variation aware architecture. In *IEEE Journal of Solid-state Circuits*, 40(9):1803-1814, September, 2005.

[6] Anant Agarwal and Steven D. Pudar. Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. In *20th Annual International Symposium on Computer Architecture*, pages 179-190, 1993.

[7] Haitham Akkary, Ravi Rajwar, and Srikanth T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, pages 423, 2003.

[8] R. Iris Bahar, Gianluca Albera, Srilatha Manne. Power and performance tradeoffs using various caching strategies. In *Proceedings of the 1998 international symposium on Low power electronics and design*, pages 64-49, 1998.

[9] James Balfour, R. Curtis Harting, and William J. Dally. Operand Registers and Explicit Operand Forwarding. In *Computer Architecture Letters*, 8(2):60-63, 2009.

[10] Deniz Balkan, Joseph Sharkey, Dmitry Ponomarev, and Kanad Ghose. Selective writeback: exploiting transient values for energy-efficiency and performance. In *Proceedings of the 2006 international symposium on Low power electronics and design*, pages 37-42, 2006.

[11] L. Baugh and C. Zilles. Decomposing the load-store queue by function for power reduction and scalability. In *IBM Journal of Research and Development*, 50(2):287-297, 2006.

[12] Eric Borch, Srilatha Manne, Joel Emer, and Eric Tune. Loose Loops Sink Chips. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 299, 2002.

[13] David Brooks and Margaret Martonosi. Dynamically Exploiting Narrow Width Oper-

ands to Improve Processor Power and Performance. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, pages 13, 1999.

[14] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 83-94, 2000.

[15] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Computer Architecture News*, 25(3):13-25, 1997.

[16] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. *In Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, pages 244, 1996.

[17] Ramon Canal, Antonio Gonzalez, and James E. Smith. Very low power pipelines using significance compression. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 181-190, 2000.

[18] J.H. Chang, H. Chao, and K. So. Cache design of a sub-micron CMOS System/370. In *14th Annual International Symposium on Computer Architecture*, pages 208-213, June 1987.

[19] F. Chen. Method and Apparatus for Implementing Circular Priority Encoder. Intel Corporation, U.S. Patent 6696988 B2, November 4, 1997.

[20] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th annual international symposium on Computer architecture*, pages 142-153, 1998.

[21] Adrian Cristal, Oliverio J. Santana, Francisco Cazorla, Marco Galluzzi, Tanausu Ramirez, Miquel Pericas, and Mateo Valero. Kilo-Instruction Processors: Overcoming the Memory Wall. *IEEE Micro*, 25(3):48-57, 2005.

[22] James Dundas and Trevor Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th international conference on Supercomputing*, pages 68-75, 1997.

[23] Carl Ebeling, Darren C. Cronquist, and Paul Franklin. RaPiD - Reconfigurable Pipelined Datapath. In *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, pages 126-135, 1996.

[24] Oguz Ergin, Deniz Balkan, Kanad Ghose, and Dmitry Ponomarev. Register Packing: Exploiting Narrow-Width Operands for Reducing Register File Pressure. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 304-315, 2004.

[25] Keith I. Farkas, Paul Chow, and Norman P. Jouppi. Register File Design Considerations in Dynamically Scheduled Processors. In *HPCA '96: Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, pages 40, 1996.

[26] G. Gerosa, S. Curtis, M. D'Addeo, B. Bo Jiang Kuttanna, F. Merchant, B. Patel, M.H. Taufique, and H. Samarchi. A sub-2 W low power IA processor for mobile internet devices in 45 nm high-k metal gate CMOS. In *Journal of Solid-State Circuits*, 44(1):73-82, 2008.

[27] Kanad Ghose and Milind B. Kamble. Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In *Proceedings of the 1999 international symposium on Low power electronics and design*, pages 70-75, 1999.

[28] A. Gonzalez, J. Gonzales, and M. Valero. Virtual-Physical Registers. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 175, 1998.

[29] Isidiro Gonzalez, Marco Galluzzi, Alex Veidenbaum, Marco A. Ramirez, Adrian Cristal, and Mateo Valero. A distributed processor state management architecture for large-window processors. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 11-22, 2008.

[30] Erika Gunadi and Mikko H. Lipasti. Cache Pipelining with Partial Operand Knowledge. In *Proceedings of Workshop of Complexity-Effective Design*, June 2004.

[31] Erika Gunadi and Mikko H. Lipasti. Narrow Width Dynamic Scheduling. In *Journal of Instruction-Level Parallelism*, vol. 9, 2007 (http://www.jilp.org/vol9/)

[32] Erika Gunadi and Mikko H. Lipasti. A position-insensitive finished store buffer. *In Proceedings of the 25th International Symposium on Computer Design*, pages 105-112, 2007.

[33] S. Gurindar. Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers. In *IEEE Transaction Computers*, 39(3):349-359, 1990.

[34] Linley Gwenapp. Intel's P6 uses decoupled superscalar deisgn. In *Microprocessor Report*, 9(2):9-15, February 1995.

[35] J.R. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. In *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, pages 12, 1997.

[36] Dana S. Henry, Bradley C. Kuszmaul, Gabriel H. Loh, and Rahul Sami. Circuits for wide-window superscalar processors. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 236-247, 2000.

[37] Dana S. Henry, Bradley C. Kuszmaul, and Vinod Viswanath. The Ultrascalar Processor-An Asymptotically Scalable Superscalar Microarchitecture. In *Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*, pages 256, 1999.

[38] E.L. Hill and M.H. Lipasti. Transparent mode flip-flops for collapsible pipelines. In . *ICCD 2007: 25th International Conference on Computer Design*, pages 553-560, October 2007.

[39] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The Microarchitecture of the Pentium 4 Processor. In *Intel Technology Journal Q1*, 2001.

[40] R. Ho, K.W. Mai, and M.A. Horowitz. The future of wires. In *Proceedings of the IEEE*, 89(4):490-504, April 2001.

[41] Mark Horowitz, Elad Alon, and Dinesh Patil. Scaling, Power, and the Future of CMOS. In *Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference: Embedded Systems*, pages 23, 2007.

[42] Koji Inoue, Tohru Ishihara, Kazuaki Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *Proceedings of the 1999 international symposium on Low power electronics and design*, pages 273-275, 1999.

[43] H.M. Jacobson. Improved clock-gating through transparent pipelining. In *ISLPED '04. Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, pages 26 - 31, August 2004.

[44] Stephen Jourdan, Ronny Ronen, Michael Bekerman, Bishara Shomar, and Adi Yoaz. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 216-225, 1998.

[45] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. In *IBM Journal of Research and Development*, 49(4/5):589-604, July 2005.

[46] R.E. Kessler, Richard Jooss, Alvin Lebeck, and Mark D. Hill. Inexpensive implementations of set-associativity. In *16th Annual International Symposium on Computer Architecture*, May 1989.

[47] R.E. Kessler, E.J. McLellan, and D.A. Webb. The Alpha 21264 Microprocessor Architecture. In *ICCD '98: Proceedings of the International Conference on Computer Design*, pages 90, Washington, DC, USA, 1998. IEEE Computer Society.

[48] Ilhyun Kim and Mikko H. Lipasti. Half-price architecture. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 28-38, 2003.

[49] Nam Sung Kim and Trevor Mudge. Reducing register ports using delayed write-back queues and operand pre-fetch. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 172-182, 2003.

[50] Soontae Kim, N. Vijaykrishnan, M.J. Irwin, L.K. John. On load latency in low-power. In *Proceedings of the 2003 international symposium on Low power electronics and design*, pages 258-261, 2003.

[51] Soontae Kim, N. Vijaykrishnan, M.J. Irwin. Reducing non-deterministic loads in low-power caches via early cache set resolution. In *Microprocessors & Microsystems*, 31(5):293-301, 2007.

[52] Johnson Kin, Munish Gupta, and William H. Mangione-smith. The Filter Cache: An Energy Efficient Memory Structure. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 184-193, 1997.

[53] Masaaki Kondo and Hiroshi Nakamura. A Small, Fast and Low-Power Register File by Bit-Partitioning. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 40-49, 2005.

[54] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. In *IEEE Micro*, 25(2):21-29, March 2005.

[55] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. In *ACM SIGOPS Operating Systems Review*, 32(5):46-57, 1998.

[56] Mikko H. Lipasti, Brian R. Mestan, and Erika Gunadi. Physical Register Inlining. In *Proceedings of the 31st annual international symposium on Computer architecture*, pages 325, 2004.

[57] Lishing Liu. Cache designs with partial address matching. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 128-136, 1994.

[58] Jose F. Martinez, Jose Renau, Michael C. Huang, Milos Prvulovic, and Josep Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-Order Microprocessors. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, 2002.

[59] Stephen Melvin and Yale Patt. Enhancing instruction scheduling with a block-structured ISA. In *International Journal of Parallel Programming*, 23(3):221-243, 1995.

[60] Francisco J. Mesa-Martinez, Michael C. Huang, Jose Renau. SEED: scalable, efficient enforcement of dependences. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 254-264, 2006.

[61] Brian Mestan and Mikko H. Lipasti. Exploiting Partial Operand Knowledge. In *ICPP-03: Proceedings of the International Conference on Parallel Processing*, pages 369-378, 2003.

[62] Teresa Monreal, Antonio Gonzalez, Mateo Valero, Jose Gonzalez, and Victos Vinals. Delaying physical register allocation through virtual-physical registers. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 186-192, 1999.

[63] Mayan Moudgill, Keshav Pingali, and Stamatis Vassiliadis. Register renaming and dynamic speculation: an alternative approach. In *Proceedings of the 26th annual international symposium on Microarchitecture*, pages 202-213, 1993.

[64] Onur Mutlu, Hyeesoon Kim, and Yale N. Patt. Techniques for Efficient Processing in Runahead Execution Engines. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 370-381, 2005.

[65] Ramadass Nagarajan, Karthikeyan Sankaralingam, Doug Burger, and Stephen W. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 40-51, 2001.

[66] David W. Oehmke, Nathan L. Binkert, Trevor Mudge, and Steven K. Reinhardt. How to Fake 1000 Registers. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 7-18, 2005.

[67] Subbarao Palacharla, Norman P. Jouppi, and J.E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th annual international symposium on Computer architecture*, pages 206-218, 1997.

[68] Il Park, Michael D. Powell, and T.N. Vijaykumar. Reducing register ports for higher speed and lower energy. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 171-182, 2002.

[69] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. Pinpointing Representative Portions of Large Intel® Itanium® Programs with Dynamic Instrumentation. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 81-92, 2004.

[70] Aashish Phansalkar, Ajay Joshi, and Lizy K. John. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 412-423, 2007.

[71] Michael D. Powell, Amit Agarwal, T.N. Vijaykumar, Babak Falsafi, and Kaushik Roy. Reducing set-associative cache energy via way-prediction and selective direct-

mapping. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 54-65, 2001.

[72] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *ACM SIGARCH Computer Architecture News*, 31(2):422-433, 2003.

[73] Joseph J. Sharkey, Dmitry V. Ponomarev, Kanad Ghose, and Oguz Ergin. Instruction packing: reducing power and delay of the dynamic scheduling logic. In *Proceedings of the 2005 international symposium on Low power electronics and design*, pages 30-35, 2005.

[74] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3-14, 2001.

[75] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45-57, 2002.

[76] Anand Lal Shimpi. Nehalem - Everything You Need to Know about Intel's New Architecture. *http://www.anandtech.com/cpuchipsets/intel/showdoc.aspx?i=3382*.

[77] G.S. Sohi and S. Vajapeyam. Instruction issue logic for high-performance, interruptable pipelined processors. In *Proceedings of the 14th annual international symposium on Computer architecture*, pages 27-34, 1987.

[78] Srikanth T. Srinivasan, Ravi Rajwar, Haitham Akkary, Amit Gandhi, and Mike Upton. Continual flow pipelines. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 107-119, 2004.

[79] Ching-Long Su and Alvin M. Despain. Cache designs for energy efficiency. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, pages 306, 1995.

[80] Ching-Long Su and Alvin M. Despain. Cache designs trade-offs for power and performance optimization: a case study. In *Proceedings of the 1995 international symposium on Low power design*, pages 63-68, 1995.

[81] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitec-*

ture, pages 291, 2003.

[82] Steven Swanson, Andrew Schwerin, Martha Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, Susan J. Eggers. The WaveScalar architecture. In *ACM Transaction on Computer System*, 25(2):4, 2007.

[83] J.M. Tendler, J.S. Dodson, D.S. Fields Jr., H. Le, and B. Sinharoy. POWER4 System Microarchitecture. In *IBM Journal of Research and Development*, 46(1):5-25, January 2002.

[84] S. Thoziyoor, N. Muralimahonar, and N.P. Jouppi. CACTI 5.0. In *HPL-2007-167*, 2007.

[85] R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. In *Instruction-level parallel processors*, pages 13-21. 1995.

[86] Jessica H. Tseng and Krste Asanovic. Banked multiported register files for high-frequency superscalar microprocessors. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 62-71, 2003.

[87] Rajesh Vivekanandham, Bharadwaj Amrutur, R. Govindarajan. A scalable low power issue queue for large instruction window processors. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 167-176, 2006.

[88] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, Anant Agarwal. Baring It All to Software: Raw Machines. In *Computer*, 30(9):86-93, 1997.

[89] Chris Wilkerson, Hongliang Gao, Alaa R. Alameldeen, Zeshan Chishti, Muhammad Khellah, and Shih-Lien Lu. Trading off cache capacity for reliability to enable low voltage operation. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 203-214, 2008.

[90] Emmett Witchel, Sam Larsen, C. Scott Ananian, and Krste Asanovic. Direct addressed caches for reduced power consumption. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 124-133, 2001.