Virtual Global Communication

by

Daniel J. Ernst

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2005

Doctoral Committee:
Todd M. Austin, Chair
Trevor Mudge
David Blaauw
Dennis Sylvester

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Todd Austin. His unparalleled creativity helped spawn many of these ideas, and his boundless enthusiasm was a driving force in getting them investigated. Since taking me as his student, his good real-world advice and his endless patience have helped me develop into a better researcher and a better person.

In addition, I thank the other members of my dissertation committee: Trevor, David, and Dennis. Your comments, suggestions, and ideas have solidified much of this work in very concrete ways. During my time as a graduate student, you've all greatly aided my understanding of a great many topics that I came asking about.

Over the course of my 5 years at Michigan, I've had the pleasure of sharing projects, meetings, and offices with many other very talented people. In particular, I would like to thank Andrew Hamel for his hard work on Cyclone and the many outstanding students and faculty who put their efforts into the Razor project and prototype. Also, I thank Chris

Weaver, Matt Guthaus, and Shidhartha Das for graciously putting up with my endless requests for help with the pleathora of CAD tools used in my studies. Of course, the culture of an office goes far beyond work, and I'm extremely grateful to have had people like Eric Larson, Chris Weaver, Rajeev Krishna, Pat Cassleman, Steve Raasch, Dave Greene, Fadi Gebara, Steve Martin, Leyla Nazhandali, and countless others around to discuss important topics like football, video games, and the latest conference gossip.

I have had the fortune to receive funding from several generous organizations, including Intel, the NSF, GSRC, and DARPA. Their support played an important role in the completion of this work.

I thank my parents, Charles and Patricia Ernst. Their dedication to education and to their family has instilled in me an understanding of the tremendous value of both, and I have yet to find a better set of mentors in either.

Finally, I thank my wife, Beth. Her love and support has been unwavering and invaluable throughout this process. Yes, you can finally stop asking, "are you done yet?"

Table of Contents

Acknowledgements ii
List of Figures vi
List of Tables viii
Abstract ix
CHAPTER 1 : INTRODUCTION 1
1.1 Motivation and Goals 1
1.2 Dynamic Scheduling 4
1.3 Control Signals 5
1.4 Contributions 6
1.5 Overview of the Thesis 8
CHAPTER 2 : BACKGROUND AND RELATED WORK 9
2.1 High-Performance Dynamic Scheduling 10
2.2 Long Wires 18
CHAPTER 3 : EVALUATION METHODOLOGY 24
3.1 Architectural Evaluation 24
3.2 Circuit Analysis Methodology 27
3.3 Area Estimate Methodology 31
3.4 Evaluating Trade-Offs 32
CHAPTER 4 : REDUCING COMPLEXITY IN DYNAMIC INSTRUCTION SCHEDULING 35
4.1 Tag Elimination 35
4.2 Cyclone 64
4.3 Summary 86
CHAPTER 5 : VIRTUAL GLOBAL CONTROL 88
5.1 Sending Control Signals Over Multiple Cycles 88
5.2 Applying Virtual Global Control to the Razor Prototype Chip 105
5.3 Summary 117
CHAPTER 6 : CONCLUSION 118
6.1 Summary of the Thesis 118

6.2 Future Directions 120
BIBLIOGRAPHY 123

List of Figures

Figure 1. Delay cost for increasing instruction window size for a 4-issue processor .2
Figure 2. Reachable Chip Area by Process Generation (taken from Matzke [45]) 3
Figure 3. Conventional Dynamic Scheduler Pipeline Stages 10
Figure 4. Reservation Station Datapaths and Control 11
Figure 5. Diagram of relevant parameters in calculating wire delay 19
Figure 6. IPns Calculation with Clock Cap 33
Figure 7. Runtime Distribution of Ready Input Operands 36
Figure 8. Conventional and Reduced-Tag Reservation Stations 38
Figure 9. GSHARE-Style Last Tag Predictor 40
Figure 10. Prediction Accuracy of Last Tag Predictors of Various Sizes 41
Figure 11. Scheduler Pipeline with Last Tag Speculation.....42
Figure 12. Reduced-Tag Reservation Station with Last Tag Speculation 43
Figure 13. Half-Price Selective Replay Mechanism. (Figure from [38]).....45
Figure 14. Intel Selective Replay Mechanism. 48
Figure 15. Instructions Per Cycle for Varying Configurations.....51
Figure 16. Scheduling delays for various tag elimination configurations.....53
Figure 17. Instructions Per ns for Varying Configurations 55
Figure 18. Impact of Tag-Reduction for Varying Window Sizes 56
Figure 19. Energy-Delay Product for Varying Configurations 57
Figure 20. Effect of Selective Replay on Reduced-Tag Schedulers.....59
Figure 21. Effect of Reduced Scheduler Pressure 61
Figure 22. Compile-time vs. Run-time Instruction Scheduling 65
Figure 23. Cyclone Scheduler Architecture 68
Figure 24. Pre-scheduler Design and Example 70
Figure 25. Switchback Logic 75
Figure 26. IPC effects of Cyclone optimizations 81
Figure 27. Performance and area for the 4-wide scheduler design space 84
Figure 28. Performance and area for the 8-wide scheduler design space 85
Figure 29. Performance and area overview.....86
Figure 30. Pipeline stalling using global clock gating 91
Figure 31. Timing diagram for pipelined stalls.....91
Figure 32. “Off-Ramps” for delayed stall signals 91
Figure 33. Mechanism for VGC micro-rollback.....93

Figure 34. Rollback register design 1 95
Figure 35. Rollback register design 2 95
Figure 36. Pipeline recovery using counterflow pipelining 100
Figure 37. Pipeline augmented with Razor latches and control lines. 105
Figure 38. Timing example of Razor operation 106
Figure 39. Razor prototype chip and vital statistics 108
Figure 40. Relative IPC for Various Razor Error Signal Designs 110
Figure 41. Relative Energy Consumption for Various Razor Error Signal Designs . 112
Figure 42. Relative IPC for various branch flush penalties with not taken branch prediction 115
Figure 43. Relative IPC for various branch flush penalties with advanced branch prediction 115
Figure 44. Relative IPns for various branch flush penalties with not taken branch prediction 116
Figure 45. Relative IPns for various branch flush penalties with advanced branch prediction 116
Figure 46. Broadcast-free dynamic scheduling element breakdown 120

List of Tables

Table 1. Key SPICE device characteristics.....29
Table 2. Circuit Characteristics of Studied Scheduler Configurations. 53
Table 3. Critical path latencies calculated with SPICE for different scheduler configurations 83
Table 4. Component breakdown for scheduler and register file area 83
Table 5. Power Consumption of registers with micro rollback 97

ABSTRACT

Virtual Global Communication

by

Daniel J. Ernst

Chair: Todd M. Austin

As process technologies continue to improve, the number of transistors a designer can put on a chip is growing tremendously. Many techniques have been proposed by architects to use this extra space. However, many of the approaches make heavily-loaded broadcasts and long-distance wires a necessity. With the continued march into deep sub-micron (DSM) technologies, the challenge to architects is to design systems that operate efficiently in a world where global communication is expensive, due to both slower circuits and higher power consumption.

In response to this challenge, many architecture researchers have abandoned traditional architectures in an attempt to create new design paradigms from scratch that are based on only local communication. While these ideas have the possibility of avoiding

communication complexity almost completely, they come at the momentous cost of redesigning every aspect of the computer, from chip units to compiler.

The goal of this work is to find and evaluate complexity-effective alternatives to global communication mechanisms in current-generation processors. Secondly, it is my goal to design these alternatives in such a way that they do not radically change the basic architecture of the machine. I call this design goal *Virtual Global Communication*.

In this dissertation, I examine several different specific communication bottlenecks. I then present possible solutions to reduce their communication complexity, remove them from critical timing paths, and reduce power consumption.

I first explore the design space of dynamic instruction scheduling logic. I present a tag elimination mechanism and the Cyclone scheduler as methods for reducing and removing complex broadcast signals, respectively. Tag elimination is shown to reduce broadcast loading by up to 75%, while Cyclone is shown to provide competitive performance with a very fast broadcast-free circuit on a small chip footprint.

I also study the complexity of some high-complexity global control signals. Micro rollback and counterflow flush techniques are presented to provide mechanisms for allowing global signals to be pipelined effectively. Counterflow flush is shown to be a low-overhead method of performing flushes using only local or constrained signals. Finally, I show that micro rollback is a viable architectural method to allow stall signals to be pipelined, with only a 2-3% power and area overhead.

CHAPTER 1

INTRODUCTION

1.1 Motivation and Goals

In an effort to increase the performance of microprocessors, designers have taken several different approaches. Processors are being made wider and more speculative to take advantage of more instruction-level parallelism (ILP). Pipelines are also being made deeper [26][32][70] to maximize the frequency at which chips can be run. These improvements, largely enabled by advances in process technology, have kept alive the widely-known “Moore’s Law” of processor advancement.

However, all of these innovations have made processors much more complicated. ILP improvements come with complex mechanisms, such as those used for prediction recovery. Deeper pipelining exposes large and slow structures, which often need to be broken up in non-intuitive and inefficient ways.

In addition, these two advancement directions may work against each other. If the ILP improvement needs a large structure, it may cause difficulties in pipelining. Alternatives, to increase pipeline depth, many structures that work for ILP must be broken up or reduced in size.

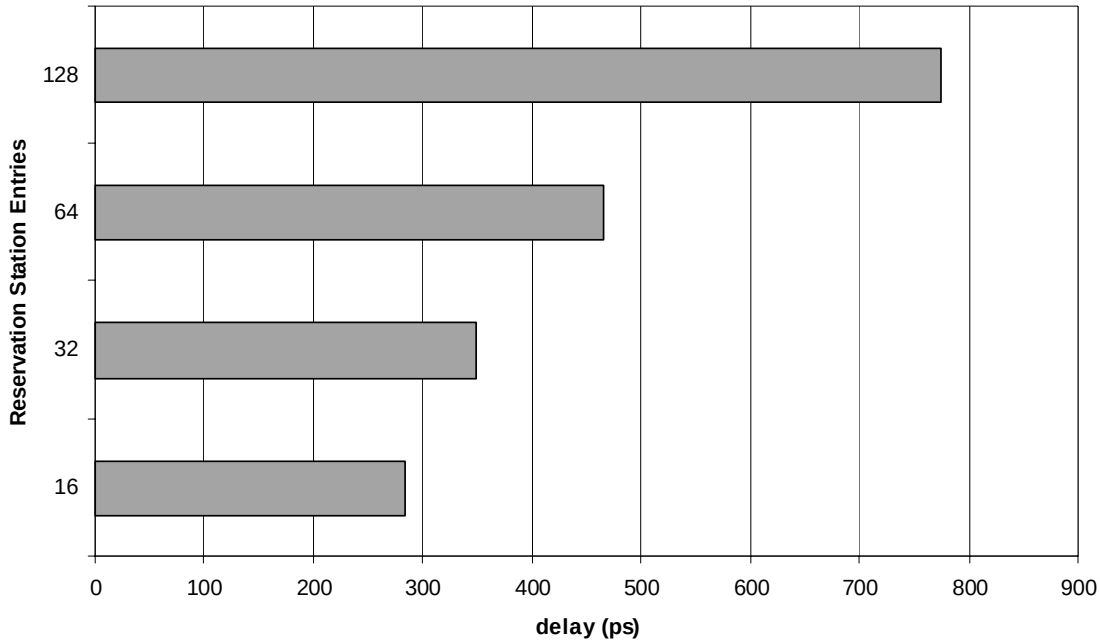


Figure 1. Delay cost for increasing instruction window size for a 4-issue processor

In dynamic instruction scheduling, for example, a simple ILP improvement can be gained by increasing the number of reservation stations available. Figure 1 shows, however, that adding more of these entries increases the delay of the structure significantly, which will make it very difficult to include as a single stage in a super-pipelined design with very short clock cycles. The simplest alternative, breaking the dynamic scheduling logic into multiple cycles, greatly reduces the effectiveness of the schedule produced. As has been studied in previous work [74], this break up hurts performance by an average of 15%.

In a similar vein, increasing pipeline depth to drive up clock frequency also has an adverse effect on chip control signals. As shown in Figure 2 (borrowed from [45]), the distance a signal can travel across a chip becomes severely limited the shorter the clock cycle becomes. This distance is squeezed by the poor RC delay scaling [45] of longer wires. In very aggressive designs, it is possible that the signal's time-of-flight [79] (its velocity

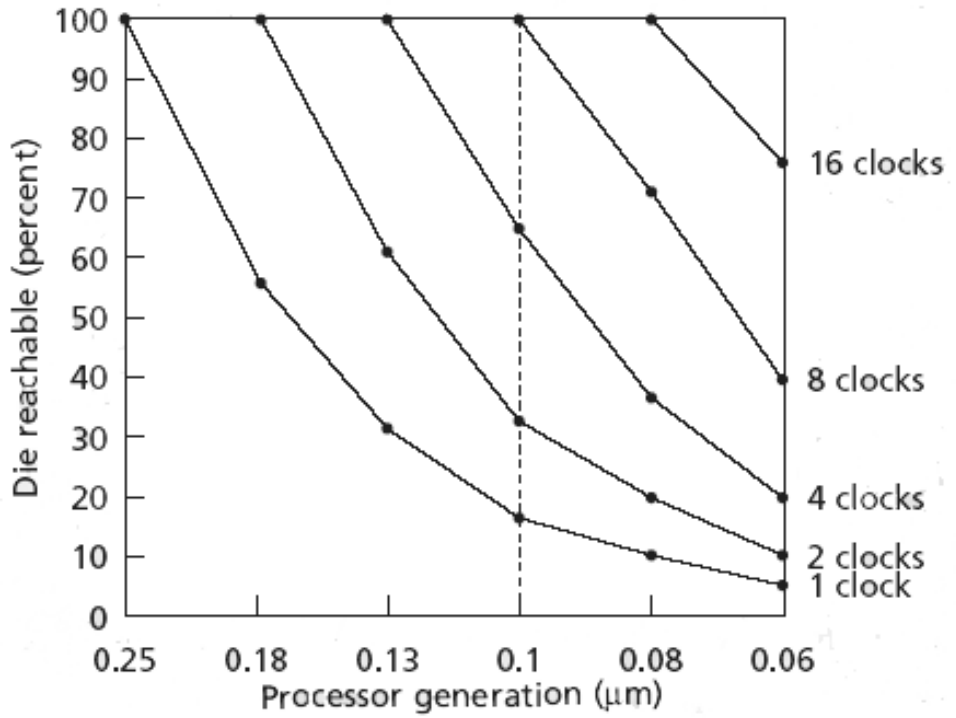


Figure 2. Reachable Chip Area by Process Generation (*taken from Matzke* [45])

through the material) may be a factor as well. In addition, many of these long signals carry specific challenges to pipelining that may result in steep penalties.

The increased complexity of processors creates several problems, including lower chip yield, longer design time, and a much more difficult verification effort. One of the biggest complexity problems is *communication complexity*, which brings about both slower circuits and increased power consumption as processes shrink and structures grow.

Communication complexity comes from this need for certain chip signals to travel long distances and communicate with a large number of recipients. These types of signals can cause problems because their highly capacitive loads are difficult to drive quickly and they require large amounts of energy to transition in a timely manner.

In addition, scaling improvements in process technology have introduced conditions that exacerbate the communication complexity problem. In particular, the delay of wire

elements, which used to be insignificant compared to logic delays, is not improving at the same rate as logic. Large structures and broadcast signals are becoming exponentially slower due to increasing wire capacitance. This trend towards wire-delay dominance has been noted by several researchers. [2][59] As further evidence, a modern system, the Pentium 4, already has pipeline stages just for wire delays. Because of the scaling effects, it now takes a much longer time and more power to send a signal across several stages of a chip, or to a large group of recipients.

The longest wires are often seen in global processor control mechanisms, such as speculation recovery, dynamic scheduling, and clocking. These instances of communication complexity usually occur where signals need to be sent a long distance across multiple stages, or a signal has multiple recipient circuits (and therefore, a highly capacitive load).

1.2 Dynamic Scheduling

In an effort to secure higher levels of system performance, microprocessor designs often employ dynamic scheduling as a technique to extract instruction level parallelism (ILP) from serial instruction streams. Conventional dynamic scheduler designs house a "window" of candidate instructions from which ready instructions are sent to functional units in an out-of-order data flow fashion. The instruction window is implemented using large monolithic content addressable memories (CAMs) that track instructions and their input dependencies.

While more ILP can be extracted with a larger instruction window (and accordingly larger CAM structure), this increased parallelism will come at the expense of a slower scheduler clock speed. Recent circuit level studies of dynamic scheduler logic has shown that the scheduler CAM logic will dominate the latency for the structure [59], and as such,

window sizes cannot be increased without commensurate increases in scheduler operation latency. More recent studies [2] also suggest that increasing wire latencies due to parasitic capacitance effects may make these trade-offs even more acute, with future designs seeing little benefit from smaller technologies. The optimal design is dependent on both the degree to which ILP can be harvested from the workload and the circuit characteristics of the technology used to implement the scheduler.

In addition to performance, power dissipation has become an increasing concern in the design of high-performance microprocessors. Increasing clock speeds and diminishing voltage margins have combined to produce designs that are increasingly difficult to cool. Additionally, embedded processors are more sensitive to energy usage as these designs are often powered by batteries. Empirical [23][44] and analytical [10][21] studies have shown that the scheduler logic consumes a large portion of a microprocessor's power and energy budgets, making the scheduler a prime target for power optimizations. For example, the scheduler components of the Pentium Pro microarchitcture consume 16% of total chip power. A similar study for Compaq's Alpha 21264 microprocessor found that 18% of total chip power was consumed by the scheduler. Increasing window sizes and parasitic capacitances will continue to shift more of the power budget towards the scheduler.

1.3 Control Signals

Some of the longest wire communications in processors occur in *recovery* logic, which is responsible for correcting the repercussions of mispeculations throughout the depth of the pipeline. *Flush* logic removes or invalidates instructions from the pipeline that the processor speculated on incorrectly. The most obvious example of flush logic is the pipeline flush that occurs when a branch instruction is mispredicted. In this case, all instructions

that follow the branch must be invalidated, and fetching resumed at the correct location. *Stall* logic causes instructions in the pipeline to stop their movement to the next stage so that some dependency further ahead in the pipeline can take extra time to resolve. A simple example of this is the “Load-Use” stall from the standard DLX pipeline [27].

The high complexity of recovery logic comes from the need to tell many stages of the pipeline that some event has occurred. Long wires are needed to distribute this information throughout the chip. Further, the loading at the end of these wires is not small, due to the need to send the signal to several different locations. These two factors combine to make the broadcast of recovery signals very slow.

1.4 Contributions

It is my goal to find and evaluate complexity-effective alternatives to the global control mechanisms which have high communication complexity. Secondly, it is my goal to design these alternatives in such a way that they do not radically change the basic architecture of the machine. I call this design goal *Virtual Global Communication*. In this work, I will address two different specific communication complexity problems: the high-capacitance multiple-recipient (broadcast) signals present in dynamic instruction scheduling, and the long distance wire communications necessary for pipeline stall and flush signals.

1.4.1 Lowering Complexity in Dynamic Scheduling

Dynamic instruction scheduling logic is a common instance of broadcast communication problems. First, Tag Elimination [17] is presented as a method for reducing the load on high-capacitance scheduler wakeup broadcast busses. This technique takes advantage of the fact that most scheduler bus entries are wasted on operands that are no longer in

flight. Using Tag Elimination, the capacitance of the wakeup bus can be reduced up to 75%, resulting in large benefits for both speed and energy consumption.

Second, the Cyclone scheduler [18], a broadcast-free scheduler based on latency prediction, is presented. This design eschews broadcast busses and instead relies upon neighbor-to-neighbor communication between instructions for routing them into their correct issue slot. This approach also includes in-line support for memory scheduling and selective scheduler replay. The Cyclone approach, while reducing IPC of the processor somewhat, also allows much higher clock speeds and uses far less area.

1.4.2 Lowering Complexity in Global Control Signals

To address some common long wire problems, I demonstrate architectural methods to pipeline various different control signals. First, I present micro rollback as a method to break the single-cycle constraint on processor stall and error signals. By retaining some processor state for an extra cycle or two, this technique can allow long distance stall signals extra time to propagate across a chip, while keeping the general operation of the processor the same.

Second, I describe a counterflow-style flushing mechanism, which streamlines recovery from processor mispredictions. Instead of requiring flush signals to travel across a chip immediately, this method allows the signal to propagate stage-by-stage through the chip, reducing timing constraints.

The culmination of all of these types of optimizations is a concept I call *Virtual Global Control*. The idea behind VGC is for a design to emulate the same operation as a processor

that includes global control signals, but doing it with only shorter module-to-module connections.

The advantage of this kind of design would be two-fold. First, the long, slow, power hungry wires can be removed. Second, it removes those elements while keeping the processor design logically very similar to a baseline processor. It can therefore be very simple to include in a design, and would avoid the problems introduced by changing the architecture drastically, as is done in projects such as TRIPS [56][64] or WaveScalar [75].

1.5 Overview of the Thesis

This thesis contains six chapters. In Chapter 2, I will give background information related to dynamic scheduling and low-complexity mechanisms, and will also discuss relevant related work. In Chapter 3, I will describe the methodologies used to examine the wide array of factors encountered in this design space. In Chapter 4, I will describe two methods of reducing communication complexity in dynamic scheduling, tag elimination and the Cyclone design. In Chapter 5, I will describe two techniques for removing long single-cycle control wires in recovery mechanisms, counterflow flush and micro rollback. Finally, Chapter 6 summarizes the thesis and provides directions for future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

The study of communication complexity is not a new one. In 1997, a study [59][60] by Palacharla, Jouppi, and Smith examined the effect of scaled fabrication processes on the cost of communication. Their conclusion was that long wires and large loads were going to become much more of a design factor than gate delays as processes dropped past 0.25µ and into the “deep sub-micron” range. In 2000, Agarwal et al. [2] studied the scaling effects that architects will have to address as future sub-micron technologies take over. One of their largest findings was that large increases in wire resistance and capacitance would make current-era micro-architectures infeasible due to a slowdown on long communication paths. The impact of global wire scaling has also been the subject of several other papers [78][31].

The processor stage that Palacharla, et al. highlighted as a primary bottleneck exacerbated by these scaling issues was the dynamic instruction scheduler. In this Chapter, I will discuss the standard implementation of this structure and the problems created by its nature, and examine work done by others to address these problems.

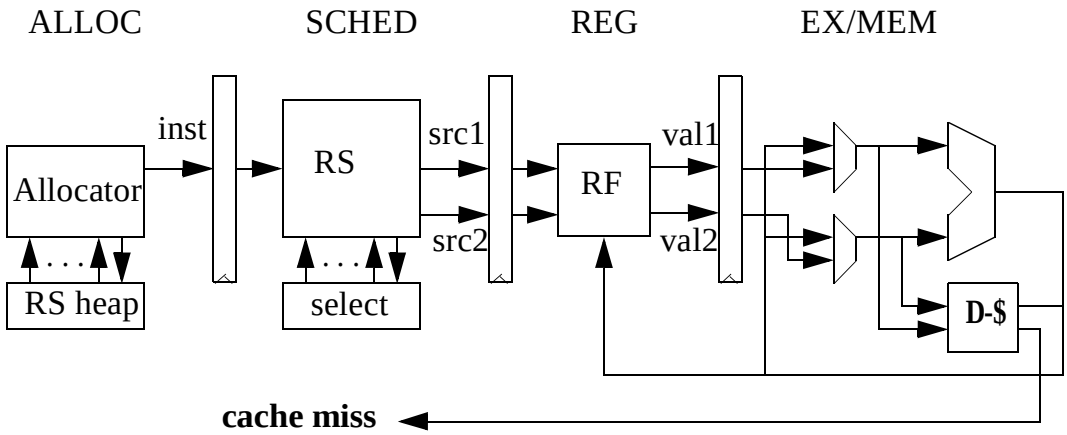


Figure 3. Conventional Dynamic Scheduler Pipeline Stages

As Agarwal and others have discussed, long communication paths scale poorly into smaller technologies. Later in this Chapter, I will discuss some of the long paths that fall into this category, and mention some current proposed solutions.

2.1 High-Performance Dynamic Scheduling

2.1.1 Scheduler Pipeline Overview

Figure 3 details the pipeline stages used to implement a high performance dynamic scheduler. The first stage, the allocator (ALLOC), is responsible for reserving all resources necessary to house an instruction in the processor instruction window. These resources include reservation stations, re-order buffer entries, and physical registers. Physical registers and re-order buffer entries typically use a FIFO allocation strategy in which the resources are allocated from a circular hardware queue.[1] This approach works well because resources are allocated in program order in ALLOC and held until instruction

1. While it is conceivable that physical registers could delay allocation until writeback, *i.e.* when the resource is needed to store the result, most designs avoid any type of out-of-order allocation because it introduces many deadlock scenarios. A good treatment on out-of-order register allocation and its potential hazards can be found in [22].

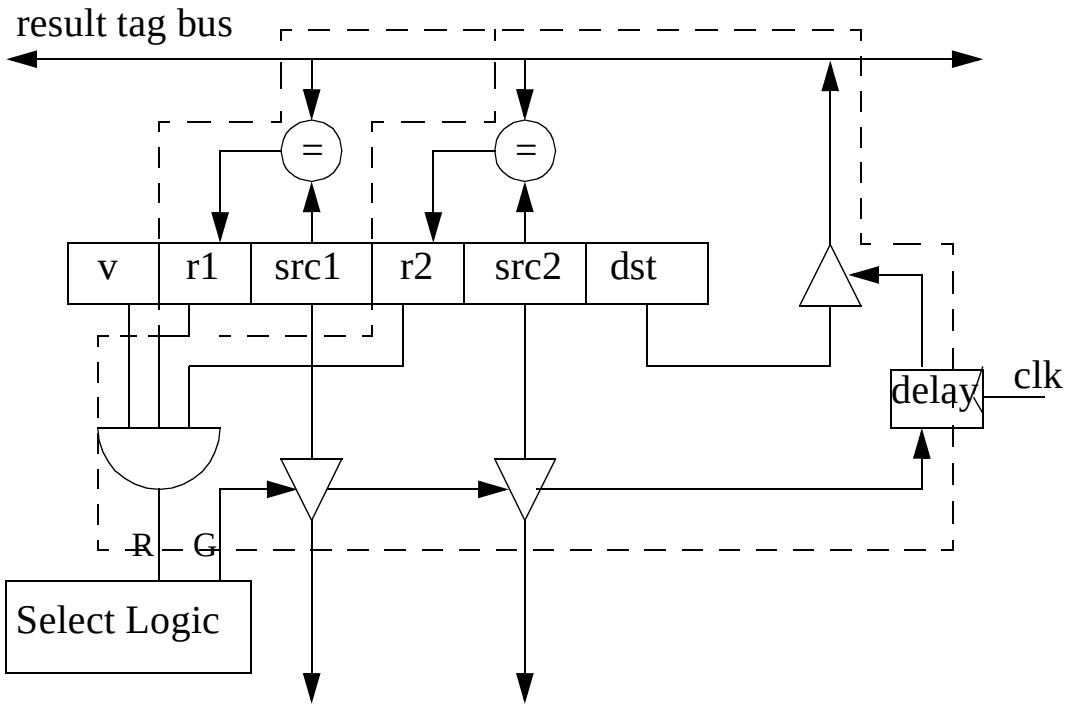


Figure 4. Reservation Station Datapaths and Control. Critical path shown with dashed line.

commit time, where the resources are released in program order. Reservation stations, while allocated in program order, may be released as soon as an instruction has begun execution (or as soon as an instruction has begun execution for the last time in a pipeline with replay/re-execution support). As such, a heap-style allocation strategy will result in more efficient use of reservation station resources than can be attained using a FIFO allocation policy.

The scheduler stage (SCHED) houses instructions in reservation stations until they are ready to execute. Reservation stations track the availability of instruction source operands. When all input operands are available, a request is made to the select logic for execution. The selection logic chooses the next instructions to execute from all ready instructions, based on the scheduler policy. The selected instructions receive a grant signal from the selection logic, at which point they will be sent forward to later stages in the pipeline.

Once granted execution, an instruction's source register tags are used to access the register file in the register read (REG) stage of the pipeline. In the following stage, operand values read from the register file are forwarded to the appropriate functional unit in the execute stage (EX/MEM) of the pipeline. If a dependent operation immediately follows an instruction, it will read a stale value from the physical register file. A bypass MUX is provided in the EX/MEM stage to will select between the incoming register operand, or a more recent value on the bypass bus. Dependent instructions that execute in subsequent cycles must communicate via the bypass bus. All other instructions communicate by way of the physical register file.

2.1.2 Reservation Stations

Figure 4 illustrates the datapaths and control logic contained in each reservation station. Each new instruction is placed into a reservation station by the allocator. If an instruction's input operand has already been computed (or if the operand is not used by the instruction), the ready bit for that operand is set as valid. If the operand has not yet been computed, a unique tag for the value is placed into the corresponding source operand tag field, either *src1* or *src2* depending on which instruction operand is being processed. Since all input operands are renamed to physical storage, the physical register index suffices as a unique *tag* for each value in flight within the instruction window. Unlike most textbook descriptions of Tomasulo's algorithm [27], most modern processors, such as the Alpha 21264 [35] and Pentium 4, use value-less reservation stations. Instead of storing instruction operand values and opcodes in the CAM structure and making longer tag busses, these designs keep this data in the REG stage where it can be accessed on the way to execution.

When instructions are nearing the completion of their execution, they broadcast their result tag onto the result tag bus. Reservation stations snoop the result tag bus, waiting for a tag to appear that matches either of their source operand tags. If a match is found, the ready bit of the matching operand tag is set. When a valid reservation station has both operands marked ready, a request for execution is sent to the selection logic. The selection logic grants the execution request if the appropriate functional unit is available and the requesting instruction has the highest priority among instructions that are ready to execute. Policies for determining the highest priority instruction vary. Some proposed approaches include random [59], oldest-first [59], and highest-latency first [73]. However, more capable schedulers require more complex logic and thus run more slowly.

The selection logic sends an instruction to execution by driving its grant signal. The input operand tags are driven onto an output bus where they are latched for use by the REG stage in the following cycle. In addition, the grant signal is latched at the reservation station. In the following cycle, the instruction will drive its result tag on to the result tag bus. If the execution pipeline supports multi-cycle operations, the result tag broadcast must be delayed until the instruction result is produced. This can be implemented by inserting a small delay element into the grant latch, such as a small counter. If the execution time for an instruction is non-deterministic, such as for a memory operation, the scheduler can optimistically predict that the latency will be the most common case, *e.g.* it predicts that all loads will hit in the data cache. If an instruction's latency is mispredicted, *e.g.* a load missed in the cache, dependent instructions were scheduled too soon and must be rescheduled to execute after the operation completes. This rescheduling is sometimes called a scheduler replay [84].

The reservation station wakeup and select logic forms the control critical path in the dynamically scheduled pipeline [59]. This logic forms a critical speed path in most aggressive designs because it limits the rate at which instructions can begin execution. As shown by the dashed lines in Figure 4, the scheduler critical path includes the result tag driver, the result tag bus interconnect, the reservation station comparators, the selection logic, and the grant signal interconnect. It is possible that the operand tag output busses (src1 and src2) are on the critical path of the control loop, however, in aggressive designs this output can be pipelined or wave-pipelined [58] into subsequent scheduler cycles because the output bus value is not required to initiate the next scheduler loop iteration. As noted by Palacharla et al [59], the CAM structure formed by the result tag drivers, result tag bus, and comparators constitute the major portion of the control circuit latency, especially for large windows with many reservation stations.

2.1.3 Reducing Broadcast Structures in Dynamic Scheduling

There have been several efforts to reduce the complexity of dynamic schedulers, which has traditionally been a very high-complexity processor component due to expensive signal broadcasts.

Some current designs bank their selection logic by having separate groups of reservation stations for each group of functional units. Each of these station groups has its own, smaller, selection network. While result tag broadcasts still need to be sent to all of the reservation stations, the latency for selecting instructions for execution can be reduced. As an added consequence of this optimization, the latency of the wakeup path makes up a higher percentage of the total scheduler delay, making it a prime candidate for implementation with wakeup-specific optimizations.

Palacharla, Jouppi, and Smith, in their study on the effects of process scaling on microprocessor design, proposed a complexity-effective superscalar processor [59][60]. Their design uses a set of FIFO queues for dynamic scheduling to reduce complexity and allow for very aggressive clocking. Instructions can only be issued from the front of the queues; instructions are steered into them using dependence information. This approach attempts to maximize the number of ready instructions at the issue boundary, while keeping the issue window small.

Stark, Brown, and Patt have proposed two methods for pipelining wakeup and selection logic, allowing for a faster clock. For their first method [74], each reservation station entry carries its own input tags along with its parent instructions' input tags in order to allow back-to-back dependent instructions to execute consecutively. They also propose speculating on which parent instruction will finish last, reducing the number of "grandparent" tags that must be stored.

Their second method, select-free logic [11], enables pipelining by allowing all instructions that wakeup to broadcast back into the window the following cycle, even though some of them may not be selected for execution.

In their studies on energy-effective issue logic, Folegnani and Gonzalez [21] also made the observation that many comparators in the instruction window are unused and unnecessary. In their low-power scheduler design, tags that are marked ready do not precharge their match lines, resulting in lower comparator power consumption. This approach dynamically reduces the power consumption of the window. However, it doesn't allow for a faster clock rate. Gonzalez and Canal [13] also propose a way to reduce the overall com-

plexity of scheduling logic by using N-Use issue scheme. Their optimization takes advantage of the observation that most instruction output values are ready only once.

Michaud and Seznec [48] proposed a method for reordering instructions as they enter the instruction window. By performing dependence analysis in a pre-schedule stage, they are able to place more usable instructions into the window, increasing its effective size.

Kucuk, *et al.* [39] propose an alternate comparator circuit to reduce energy dissipation in dynamic schedulers. Their optimization, like many of the others, could be used in combination with our own tag elimination for improved energy efficiency.

LeBeck's WIB scheduler identifies instructions dependent on long latency operations (data cache misses), and directs these operations to a secondary scheduler [40]. When the long latency operation nears completion, the dependent operations are dumped *en masse* into a small CAM-based dynamic scheduling window. Morancho used a similar approach to move dependent operations following long latency instructions out of the instruction window [50]. Unlike the WIB, they record relative instruction latencies to simplify the re-execution of operations once a valid schedule has been built. By removing the long-latency instructions from the window, these designs reduce the pressure on it, which results in needing fewer entries to maintain the same performance. We utilize a similar approach in our Cyclone [18] replay mechanism. As instructions replay, dependencies between dependent operations are maintained by their spacing in the scheduler queues. Unlike the WIB and Morancho's work, our scheduler is completely broadcast free. We pick a schedule and fully commit to it for the lifetime of the instruction, using the replay mechanism to accommodate any incorrectly scheduled instructions.

Raasch's segmented instruction queue [61] utilizes course-grained timing information to direct instructions to a sequence of small CAM-based instruction windows. As instructions near execution, they move to instruction queues closer to functional unit results. Their solution therefore allows for extremely large instruction windows, while keeping the complexity at a fairly constant value, based on the size of the smaller CAM-based windows.

Various groups have proposed *Matrix* [25] (or *One-Hot* [20]) scheduling. Instead of comparing operand "tags", these designs directly link producer and consumer instructions through a dependence vector. While this structure is more efficient for small instruction windows, it scales poorly as issue queue size grows. The area of the matrix schemes scales quadratically with window size, as opposed to linearly for CAM-based windows. The large area cost makes large matrix windows slower due to long wire distances.

The use of decentralized dependence analysis, and therefore much lower communication complexity, is the central idea of the counter-dataflow architecture [72][71]. In the counterflow pipeline, instructions and data flow in opposite directions on circular queues. When instructions that are waiting to execute pass input operands, the data needed to execute is captured by the dependent instruction. Once an instruction has captured all its operands, it continues to cycle until it locates an appropriate functional unit, at which time it leaves the queue to begin execution. While this architecture can be implemented in an extremely low-complexity manner, the lack of any actual scheduling of instructions causes a lot of instruction throughput to be lost in the chaos.

2.2 Long Wires

There have been several studies examining the effect of deep sub-micron scaling on the properties of wires. Sylvester and Keutzer did several studies [76][77][78] which concluded that, while local intra-module wires should scale well into deep sub micron, global wires that must travel across many modules on the chip will have more difficulty. Their work was backed up by Horowitz [31], who suggested that architects look into more modular designs to remove long global wires.

The problem that these studies highlight comes down to the way that wires have typically been constructed as processes shrink. Figure 5 shows the important parameters for finding the *RC* delay of wire elements.

The delay of a wire can be estimated in a general sense by multiplying its resistance (*R*) and its capacitance (*C*). The resulting RC number is directly proportional to the delay of the wire. The *R* and *C* values can be estimated very roughly by the following formulas, given the values of the parameters shown in Figure 5.

(1) $$R = \frac{\rho L}{tW}$$

(2) $$C = 2 * \left(\frac{\varepsilon L t}{h} + \frac{\varepsilon L W}{v} \right)$$

For short wires, like those found locally within processor modules, *R* is the much smaller factor, so its scaling factor is ignorable. *C* scales down roughly constant per unit *L*, and since the smaller transistors are more tightly packed, *L* scales down proportional to *k*,

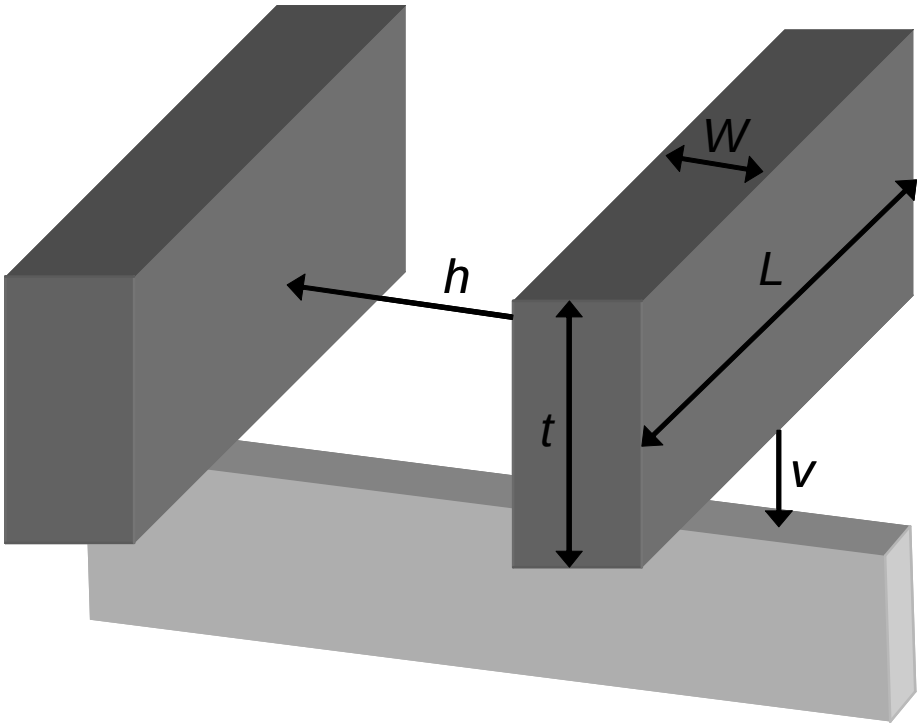


Figure 5. Diagram of relevant parameters in calculating wire delay

the overall scaling factor. Therefore, the delay of short wires within modules scale down fairly well as technologies shrink, and are not a large problem.

For long wires, however, such as those found connecting between modules or traveling across the chip, the scaling factors show a very different trend. With sufficient length L, the resistive component is no longer negligible compared to the resistance of gates. Within the *R* equation, the term *tW* represents the cross-sectional area of the wire. In the past, while *W* has always scaled smaller (to improve density), process engineers have kept *t* from also scaling down, to keep this cross-sectional area as large as possible and reduce resistance. However, as technologies have become smaller, the wire thickness *t* can no longer be maintained without severely endangering the stability of the wire (aspect ratios are already up to 2/1 and a cap is predicted at about 2.2). It is also infeasible to leave both

W and *t* large in a shrink, as *W* impacts overall chip density, and both factors have negative effects on the scaling of the capacitive element.

With the resistive component of delay scaling up at roughly k^2L, and the capacitive element scaling approximately constant with *L*, the overall scaling story for the *RC* delay appears to be that it stays approximately constant with a shrink to a smaller process, assuming *L* scales down by *k*.

If the delay of long wires stays the same in the way we have described, and the delay of transistors shrinks by *k*, this results in the wires taking up a larger portion of the stage delay than they did previously. While this appears to be a problem already, it is further exacerbated by the fact that, due to the tendency of designers to add more functionality to a shrunk design, the length *L* of a global wire is much less likely to scale down by *k*, and will likely remain a near-constant length instead. The result of this is that the delay of long wires actually would scale **up** by k^2 for a process shrink, while transistor size and speed would scale **down** by *k*.

Finally, it is worth noting that, due to its poor scaling qualities, the resistance of a wire is becoming non-negligible for smaller and smaller values of *L*. This means that more wires which were previously not major contributors to delay will move into the “long wire” category as process technologies shrink.

It is worth noting that poor delays are not the only issue with interconnect. With increased complexity comes higher dynamic power consumption from the switching of wires. A recent study by Intel [42] estimated that, in one of their production processors, 50% of dynamic power in the chip was spent by wires, clock tree excluded.

2.2.1 Reducing and Removing Wire Delays

As a result of these trends, much work has been done in efforts to reduce the latencies of these wires, or to reduce the number of long wires in processor designs.

In the process technology realm, efforts have been made to find new materials to aid in the construction of low-delay wires. As an example, the industry move from aluminum to copper wires greatly reduced their resistance, due to the lower resistivity property of copper (1.7 μΩ-cm vs. 2.8 μΩ-cm).

Besides changing the aspect ratio of wires, process technologies also now have more metal layers, allowing for wires to have a larger cross-sectional area without becoming too congested to accomplish a good routing.

At a circuit level, there are several techniques available to help reduce the delay required to send a signal long distances. Repeaters [30] can be used to reduce wire delay scaling to a linear amount per length. Using techniques such as differential wires, limited swing interconnect [29], and static pulsed buses [36] can also decrease the delay required to send signals long distances.

While all these methods aid in keeping delays manageable, none of them are a “silver bullet” solution to the problem. Some have argued [31] that it falls on chip architects to design architectures that are aware of wire limitations in scaling, and make an effort to remove long global wires.

Much work has been done by architects in an effort to reduce the number of long wires in processor design. These efforts have not usually been focused on the general problem, but attack many different structures throughout a chip individually.

In the area of memory systems, Beckman and Wood [8] proposed Transmission Line Caches. To increase speed, their design uses on-chip transmission line technology instead of long-distance global wiring.

To reduce the amount of bypass wires necessary in superscalar processors, Kim and Smith [37] proposed an architecture for Instruction-Level Distributed Processing (ILDP). Their design attempts to keep communication between clusters of a processor at a minimum by keeping register/accumulator values local to each processing element and sending instructions to the different elements based on the location of their input operands.

In the area of pipeline control, our pipeline recovery mechanism, used in the Razor design [19], is inspired from Sproull's work on asynchronous counterflow pipelines [71], which was later adapted for synchronous systems by Miller [49]. The basic idea of a counterflow pipeline is that instruction and control signals flow in a direction opposite to data values. As such, global control is not necessary as all control signals will eventually reach the appropriate point in the datapath.

On a more chip-level scale, many research groups have explored design paradigms that involve small processing elements connected to each other through only local communication nets. The TRIPS project [56][64] at UT-Austin, the Wavescalar project [75] at Washington, and the RAW project at MIT [81][86] are all examples of this kind of architectural approach. While these projects all accomplish a drastic reduction in global communication, they also bring about drastic changes in the architectural design. Much of the burden is put upon the compiler, and well-understood problems in the superscalar world, like memory ordering, can become difficult and painful to deal with.

Micro rollback was discussed at length by Tamir, Trembley, and Rennels [80][82][83] as a method to quickly recover from transient faults that occurred during execution. While this work is not directly related to low-complexity pipelines, their recovery mechanism is accomplished locally, and without a need for single-cycle long wires. I make use of their ideas in Chapter 4 in my discussion of virtual global control.

CHAPTER 3

EVALUATION METHODOLOGY

The issues discussed in this thesis have significant stakes in multiple areas of processor design. Many of the techniques have effects at the architectural level. Due to their nature, their effectiveness must also be measured by evaluating circuit timings for these designs. Finally, changes in these two areas often bring about differences in the power consumption of the circuits.

To evaluate the overall effectiveness of proposed techniques, methodologies need to be developed for each of these fields of study. In this chapter, I give details on the simulators and models used in this thesis to quantitatively examine the effects of the designs presented in the rest of this work.

3.1 Architectural Evaluation

3.1.1 Simulators

The architectural simulators used in these studies are derived from the SimpleScalar/Alpha version 3.0 tool set [6], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user-level instructions, performing a detailed timing simulation of the desired processor model. Simulation is execution-driven,

including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction.

For our dynamic scheduling studies (presented in Chapter 4), our baseline simulation configuration models a current generation out-of-order processor microarchitecture. It can fetch and issue up to 4 instructions per cycle and it has a 64 entry dynamic scheduler window with a 32 entry load/store buffer. There is a 5 cycle minimum branch misprediction penalty. The processor has 4 integer ALU units, 2-load/store units, 4-FP adders, 4-integer MULT/DIV units, and 4-FP MULT/DIV units. The latencies vary depending on the operation, but all functional units, with the exception of the divide units, are fully pipelined allowing a new instruction to initiate execution each cycle.

The memory system consists of 32k 4-way set-associative L1 instruction and data caches. The data cache is dual-ported and pipelined to allow up to two new requests each cycle. There is also a 256k 4-way set-associative L2 cache with a 6 cycle hit latency. If there is a second-level cache miss it takes a total of 80 cycles to make the round trip access to main memory.

The instruction fetch stage of the model has a 32 entry instruction queue and operates at twice the frequency as the rest of the processor. While perhaps not realistic, this assures that the IPC results seen when changing scheduler configurations are not attenuated by bottlenecks in the front end. Since improvement in scheduler performance would have to be accompanied by commensurate improvement in fetch bandwidth, we feel that this configuration will accurately portray the benefits of our scheduler optimizations.

The dynamic scheduler distributed with SimpleScalar is overly simplistic compared to modern schedulers. A redesigned scheduler was added to sim-outorder to more accurately

reflect the design points presented in Chapter 2. Our new scheduler also includes support for several varieties of scheduler replay, more efficient scheduler resource management, decoupled reorder buffer (ROB) and reservation station (RS) resources, and support for our optimizations detailed in the Chapter 4.

Both processor models use a GSHARE branch predictor with an 8-bit global history and an 8k entry BTB. As the prototype chip we based our control signal models on has only a “not taken” branch predictor, these designs were evaluated both with and without the more advanced branch predictor.

For our control signal studies (presented in Chapter 5), our baseline simulation configuration models a much different design point. The single-issue in-order processor we simulate is as close as possible to the Razor prototype chip design described in [19]. The fairly typical 5-stage pipeline was modeled in great detail, including bypass, stall events, and instruction latencies.

As the Razor test chip has only rudimentary caches (on-chip SRAMs) and no interface to an external DRAM memory, cache designs were chosen arbitrarily to portray more realistic performance. For these designs, we modeled 32k 4-way set-associative L1 instruction and data caches, with no L2. Cache misses require a total of 100 cycles to retrieve data from main the modeled main memory.

Because the Razor mechanism requires the architecture to react to dynamic data-dependant circuit evaluations, support for circuit timing is included in this simulator. Verilog models of each logic stage are simulated every cycle to determine the time it takes the underlying circuitry to complete computation for the given input vectors. Since cycle-by-cycle circuit-level simulation carries a very large computation overhead, several optimiza-

tions were applied to the circuit simulator, including early circuit simulation termination based on architectural constraints and circuit timing memoization [41].

3.1.2 Benchmarks

To perform our evaluation, we collected results from the SPEC2000 benchmarks [69]. All SPEC programs were compiled for a Compaq Alpha AXP-21264 processor using the Compaq C and Fortran compilers under the OSF/1 V4.0 operating system using full compiler optimization (-O4).

The high-performance processor we model in Chapter 4 is likely have usefulness across a broad range of applications, and so 25 integer and floating point SPEC benchmarks were run (all except *perl*, for which the input sets break our simulator). The simulations were run for 100 million instructions using the SPEC reference inputs. We used the SimPoint toolset's Early SimPoints [67] to pinpoint program locations to simulate for peak accuracy.

The processor modeled in Chapter 5 fits more closely in the high-end embedded area, so only a set of the integer SPEC benchmarks were run. Due to the very slow speed of the circuit-aware simulation used for this processor, the simulations were run for only 10 million instructions from the SPEC reference inputs. Again, the SimPoint toolset's Early SimPoints [67] provided the program locations to simulate for the highest accuracy.

3.2 Circuit Analysis Methodology

To get a full understanding of the effects of a virtual communication mechanism, simply performing architectural simulation isn't enough. The changes to the circuit structures need to be examined as part of the process of weighing all sides of the complexity picture.

For an accurate evaluation of circuitry, good process models are a necessity. For researchers without access to production processes, the Berkeley Predictive Technology Models [7] provide academic estimates for several generations of production processes. For this work, we make use of manufacturer SPICE models from both Taiwan Semiconductor Corporation (TSMC) and IBM.

To model the circuit structures in this work, we use hand-coded and optimized transistor-level SPICE netlists. Several of the netlists were further optimized using Synopsys's AMPS circuit optimization tool (version 5.5). AMPS attempts to optimize circuit latency, power, or area under a given set of constraints. We configured AMPS to optimize circuit latency, with the constraint that overall transistor area could not increase.

For simulation of the performance of these circuits, Avant!'s HSPICE circuit tool was run on the models, using level 49 typical transistor parameters supplied by TSMC for their 1.8V 0.18μm fabrication technology. Some of these parameters for this modern production-grade process are shown in Table 1. A complete list is available from MOSIS's secure website [51].

The overall power consumption of these circuits was estimated through the HSPICE simulation by examining the energy consumption for all input patterns. Average power estimates were determined by factoring in the frequency of these patterns and the overall activity rate of the unit.

3.2.1 Dynamic Scheduler Circuits

The circuit delays and power consumption statistics for scheduler windows used in this thesis were derived from an updated version of the SPICE models used in the work by

Table 1. Key SPICE device characteristics. Parameters shown are for a level 49 HSPICE model. When NMOS device characteristics differ from PMOS devices, the PMOS characteristics are shown in parenthesis. Wire capacitance values are shown given the inter-line spacing used in our scheduler design (0.478μm)

$\mathbf{V_{dd}}$	1.8 V	$\mathbf{c_j}$	0.00100 (0.00112) F/m^2
$\mathbf{v_{to}}$	0.445(-0.437) V	$\mathbf{c_{jsw}}$	2.04e-10 (2.48e-10) F/m
$\mathbf{t_{ox}}$	4.08 nm	$\mathbf{c_{jso}}$	3.66e-10 (3.28e-10) F/m
μ_o	125(100) cm^2/Vs	$\mathbf{c_{jdo}}$	3.66e-10 (3.28e-10) F/m
$\mathbf{R_{metal}}$	0.239 Ω/μm	$\mathbf{r_{sh}}$	6.8 (7.2) Ω/sq.
$\mathbf{C_{metal}}$	1.82 fF/μm with 0.478 μm spacing		

Palacharla, Jouppi, and Smith [59][60]. Palacharla's original analyses predate the existence of a functional 0.18μm fabrication technology. Because of this, the device parameters in that work were extrapolated from a Digital Equipment Corporation 0.8μm technology.

The new timing and power figures in our work are the result of porting Palacharla's original design to the TSMC production fabrication technology and performing timing optimizations using commercial tools configured for the implementation technology. AMPS provided great benefits for the design of the select circuit, producing a re-sized design that was more than 25% faster, and with only 90% of the original area. AMPS also improved wakeup latency nearly 5% with no change in area.

Overall, the ported design is about 24% faster in the commercial technology. The primary factors leading to the faster design are roughly split between faster transistor speed (due to a lower threshold voltage and gate capacitance) and improved logic performance due to better transistor sizing.

Once transistors were sized, timing analysis was performed on a SPICE representation of the optimized scheduler design, augmented with parasitic wire delays. Wire parasitics were computed in the same fashion as Palacharla's earlier study, except wire resistance and capacitance was adjusted for the TSMC production process. Finally, timing and power analysis was performed using Avant!'s HSPICE circuit tool.

3.2.2 Storage Element Analysis

In Chapter 5, we will examine the power overhead of some extra register elements needed for a virtualization mechanism. Because these elements spend a lot of time storing data without a large amount of switching, it is necessary to examine the static leakage power of the design as well as its dynamic consumption.

One process-level optimization for reducing static power is the use of high-threshold transistors [55]. Some fabrication processes provide cells and models for 2 or more different thresholds for each type of transistor. The TSMC process models used in other segments of this work do not include models for high-V_t transistor operation. However, another process available to us, the IBM 0.13µm 1.2V process, includes cell parameters for both high- and standard-V_t transistors. For this reason, the smaller and more advanced IBM process was used to evaluate the effectiveness of this specific design.

3.3 Area Estimate Methodology

3.3.1 Register Bit Equivalent

A parameter important to some designs is the chip area used by a processor element. While not many high-performance designs are strictly die-size limited, a smaller structure

footprint can also indicate lower circuit capacitance, which can translate into lower power and/or a faster clock speed.

To estimate the chip area of drastically different scheduler designs in Chapter 4, we use the process-independent register bit equivalent (RBE) metric defined by Mulder, et al. [54], where one RBE equals the area used by one register file bit. Using general equations from their scheme, the RBE size can be computed for several different standard memory circuits, such as register files, set associative caches, and fully associative CAMs. The metric takes into account both the area of the cells themselves, as well as the overhead of control logic, driver logic, and sense amps.

One parameter not accounted for in the original RBE equations given in [54] is the number of access ports for a memory structure. Because the size of each side of a memory bit must scale up linearly with the number of ports, the total effect on area is quadratic [66]. In our model for multi-ported CAM structures, we apply this port scaling factor to the portions of the RBE area equation that pertain to the footprint of the data bits.

3.3.2 Area Estimate Using Transistor Sizing

RBE estimates are not a useful tool when evaluating the area of small individual storage elements, or combinational logic circuits. To compare circuit areas between these smaller structures, we need to use a metric based on a lower-level parameter than number of storage bits. In Chapter 5, we compare different flip-flop designs which only differ in the number and sizing of transistors in the circuit. To estimate the total relative layout area, we sum the layout widths of all transistors in each design.

3.4 Evaluating Trade-Offs

Once an optimization has been simulated and its circuit changes modeled, we have two separate metrics, each effecting overall performance: instructions per cycle and clock frequency. To compact the performance of the optimization into one metric, the two must be combined.

The overall performance equation, as stated by Hennessy and Patterson in their text-book [27] is:

$$Perf = IPC \times \frac{1}{t_{clk}}$$

The resulting metric represents "*Instructions per time*" - the pure computational throughput of the machine. Historically, this metric has been given the unit MIPS (Million Instructions Per Second). However, in this thesis, we use a more up-to-date IPns (Instructions Per nanosecond), which is numerically equivalent to BIPS (Billion Instructions Per Second). While looking at the IPns value for an entire processor gives an accurate measure of total throughput, doing so on a stage-by-stage basis may represent a slightly different picture.

For example, if a specific optimization reduces stage A's circuit timing from 5 ns to 3 ns, the IPns value for that stage would be generated reflecting a 67% gain in clock speed. If stage B's longest circuit path was 4 ns long, the clock period of the entire processor, including stage A would be set at 4 ns, assuming a uniform clock period across the chip. In

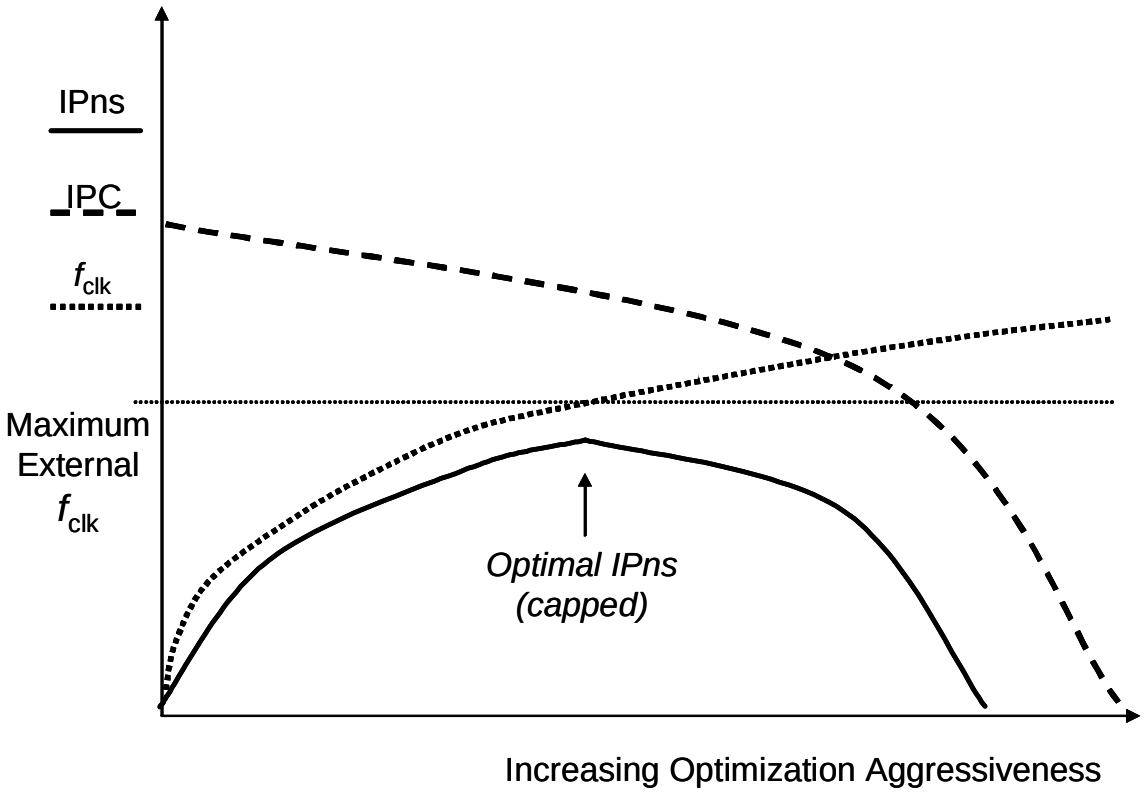


Figure 6. IPns Calculation with Clock Cap

this case, the processor-wide speedup in throughput should only reflect a 25% gain in clock speed.

This, in effect, puts a cap on possible chip-wide IPns gains due to optimizations with high circuit-speed returns. In most cases, this will mean that the optimal design point for a given stage will be the one which beats the clock speed cap and has the highest IPC. A conceptual picture of the trade-off with a capping of circuit gains is shown in Figure 6. An increase in clock frequency shows benefits in total throughput up until the cap is hit, after which the total throughput will track only the architectural IPC.

Just because further circuit speed improvements will not affect overall chip throughput does not mean that the computation of a stage-by-stage IPns gives no insights. While the global clock speed may make a higher stage IPns impossible in a literal sense, a higher

stage IPns may indicate a lower overall stage complexity. The benefits of lower-complexity circuits can be exploited in ways other than pure throughput. Given the example from earlier, the lower-complexity 3 ns circuitry for stage A could possibly be made to function at 4 ns, but using less power than a highly optimized 4 ns circuit.

Overall, the optimal mechanism for a given design will depend on the goals of the designer. While projects focused on the highest possible throughput will want to choose the optimal IPns point given the global cap, designs focused more on energy efficiency may choose to exploit additional complexity overhead given by more aggressive optimizations.

CHAPTER 4

REDUCING COMPLEXITY IN DYNAMIC INSTRUCTION SCHEDULING

In this research, we examine two different methods for reducing and removing global broadcast signals in dynamic instruction scheduling and replay. The first technique, tag elimination, drastically reduces the size and power consumption of traditional CAM-style instruction scheduling windows, while only having a minimal effect on IPC. The second approach, named “Cyclone”, replaced the scheduling window with a mechanism that removes the high complexity load of a CAM window, and relies only on local (neighbor to neighbor) communication.

4.1 Tag Elimination

In this section, we propose two scheduler tag reduction techniques that work together to improve the performance of dynamic scheduling while at the same time reducing power requirements. First, we propose a reduced-tag scheduler design that assigns instructions to reservation stations with two, one, or zero tag comparators, depending on the number of operands in flight. Secondly, to reduce tag comparison requirements for instructions with multiple operands in flight, we introduce a last tag speculation technique. This approach predicts which input operand of an instruction will arrive last, and then schedules the exe-

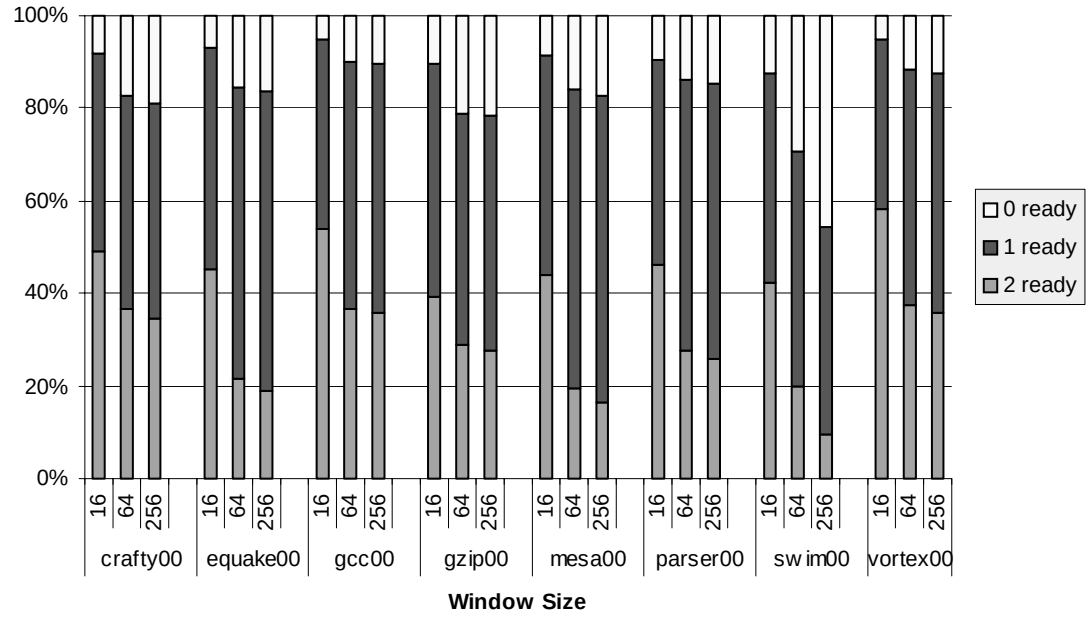


Figure 7. Runtime Distribution of Ready Input Operands

cution of that instruction based solely on the arrival of this operand. we also discuss how to integrate selective instruction replay with these mechanisms in an complexity-effective manner.

4.1.1 Specialized Windows

When an instruction enters the instruction window, its input operand tag fields are loaded with the index of the physical register that will eventually hold the operand value. It may be the case that some of the operands will be ready at that time, either because the operand was computed in an earlier cycle or the operand is not required by the operation (e.g., one of the operands is an immediate value). Since these input operands are already available, their reservation station entries do not require tag comparators.

To quantify the degree to which tag comparators are not required by reservation stations, a typical 4-wide superscalar processor was simulated using the SimpleScalar toolset

[6] with instruction window sizes of 16, 64, and 256 and an load/store queue size that was half the size of the instruction window. When instructions entered the scheduler, the number of ready input operands was counted. The results in Figure 7 show the dynamic distribution of the number of ready operands for all instructions. Results are shown for eight of the SPEC2000 benchmarks [69]. (More details on our experimental framework and baseline microarchitecture model can be found in Chapter 3.)

Clearly, a significant portion of all operands are marked ready when they enter reservation stations, for all instruction window sizes. Only about 20% of all dynamic instructions require a reservation station with two tag comparators, while the remaining instructions require either one or zero comparators. As expected, larger window sizes result in fewer ready operands, because larger windows permit the front-end to get further ahead of instruction execution. Nonetheless, a window size of 256 instructions has a significant portion of instructions that do not require more than one tag comparator. In general, programs with poor branch prediction such as *GCC* and *Vortex* were less affected by the larger windows sizes, because branch mispredictions limit the degree to which the front-end can get ahead of instruction execution. In contrast, *SWIM* has nearly perfect branch predictor accuracy, which results in more slip between fetch and execute for large window sizes. Still, even in this example, more than half of the instructions in a 256-entry instruction window require less than two tag comparators. Similar observations were made by Folegnani and Gonzalez [21]; they used this property to design low-power tag comparators.

It is possible to take advantage of ready input operands if the scheduler contains reservation stations with fewer than two tag comparators. Figure 8 illustrates a reservation sta-

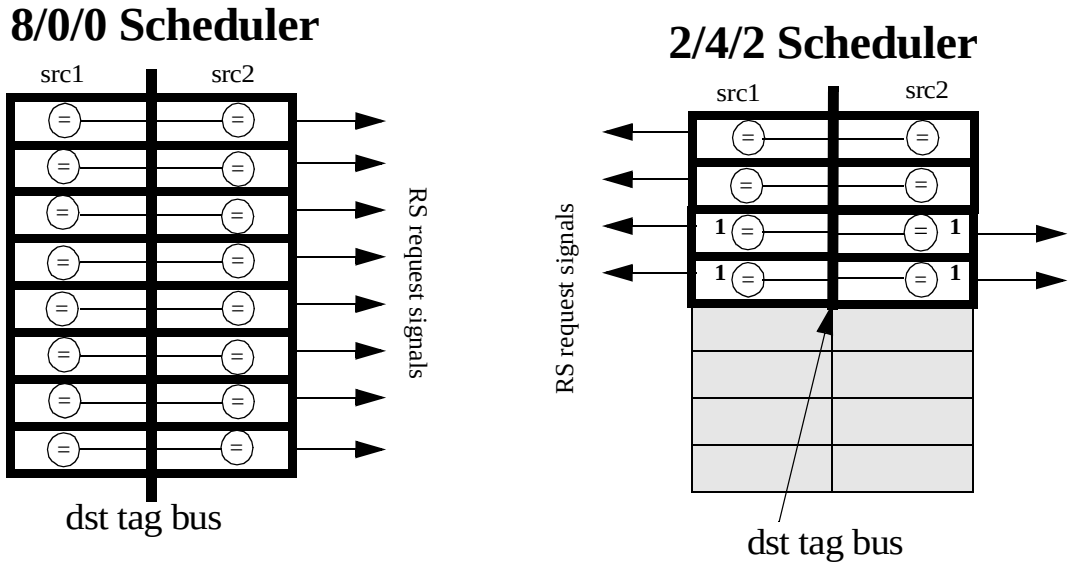


Figure 8. Conventional and Reduced-Tag Reservation Stations. The circles represent tag comparators. The bold tag entries include a comparator, the shaded tag entries are not necessary and so do not include comparators.

tion design that contains entries with two, one, and zero tag comparators. The design on the left side of the figure is a conventional scheduler configuration, where each reservation station contains two tag comparators. The optimized design, shown on the right side of the figure, eliminates tag comparators from some of the reservations stations. We label the configurations “x/y/z”, where x, y, and z indicate the number of two, one and zero tag stations, respectively.

When the allocator encounters an instruction with one or more unavailable operands, it will assign the instruction to a reservation station with a matching number of tag comparators. If both operands are ready, we can place the instruction into a reservation station without tag comparators that immediately requests execution. If there isn’t an available reservation station with the same number of tag comparators, the allocator will assign the instruction to a reservation station with more tag comparators. For example, instructions waiting for one operand can be assigned to reservation stations with one or two tag com-

parators. Finally, if a reservation station with a sufficient number of tag comparators is not available, the allocator will stall the front-end pipeline until one is available.

This reduced-tag scheduler design has two primary advantages over a conventional design. First, the destination tag bus, which drives a physical register destination tag to all source tag comparators, need only run to the reservation stations with tag comparators. Since result tag drive latency is on the critical path of the control scheduler loop, the latency of this critical path will be reduced in proportion to the number of zero-tag reservation stations. The second advantage is that comparator circuits can be eliminated from the instruction window. With fewer comparators, load capacitance on the result tag bus is reduced, resulting in faster tag drive and lower power requirements. The downside of the reduced tag design is that additional allocator stalls may be introduced when there are insufficient reservation stations of a required class, potentially reducing extracted ILP and program performance.

4.1.2 Last Tag Speculation

While many instructions enter the instruction window with multiple unavailable operands, it is still possible to eliminate all but one of the tag comparators for these instructions. Since the arrival of the all but the last input operand tags will not initiate an execution request, these tag comparators can be safely removed. When the *last* input operand tag arrives, this sole event can be used to initiate instruction execution. We employ a last-tag predictor to predict the last arriving operand. As long as the last tag predictor is correct, the schedule will proceed as in the non-speculative case. A modified reservation station with one tag comparator monitors the arrival of the predicted last input operand.

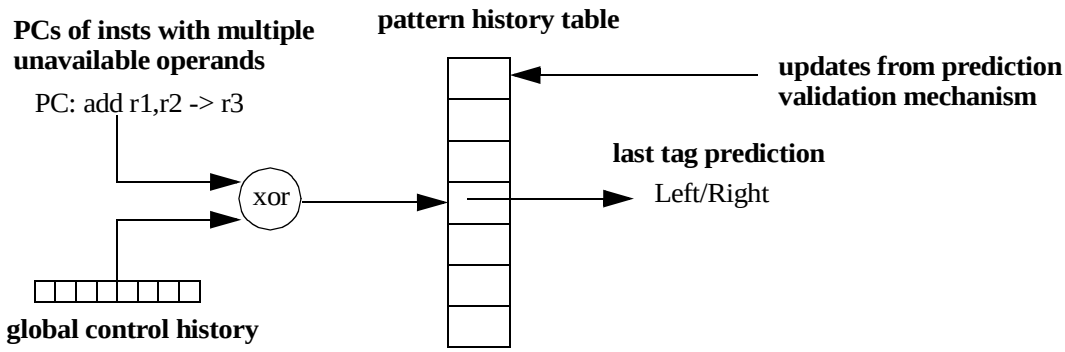


Figure 9. GSHARE-Style Last Tag Predictor

We experimented with a number of last-tag predictors, including a predictor that predicts the last operand to arrive will be the same as in the previous execution, a bimodal-type last-tag predictor (similar to the grandparent predictor employed by Stark et al [74]), and a GSHARE-style last-tag predictor. The accuracy of these predictors was nearly identical to similarly configured branch predictors. The GSHARE-style last-tag predictor consistently performed the best with only marginal additional cost over simpler predictors.

Figure 9 illustrates the GSHARE-style last-tag predictor. The predictor is indexed with the PC of an instruction (with multiple unavailable operands) hashed with global control history [46]. The control history is XOR'ed onto the least significant bits of the instruction PC and that result is used as an index into a table of two-bit saturating counters. The value of the upper counter bit indicates the prediction: one indicates the left operand will arrive last, zero indicates the right operand will arrive last. The predictors are updated when last tag predictions are validated. Figure 10 shows the prediction accuracy for various sizes of the GSHARE-style last-tag predictor with 8 bits of control history and an instruction window size of 64. Most programs have good predictor performance for sizes larger than 1024 entries.

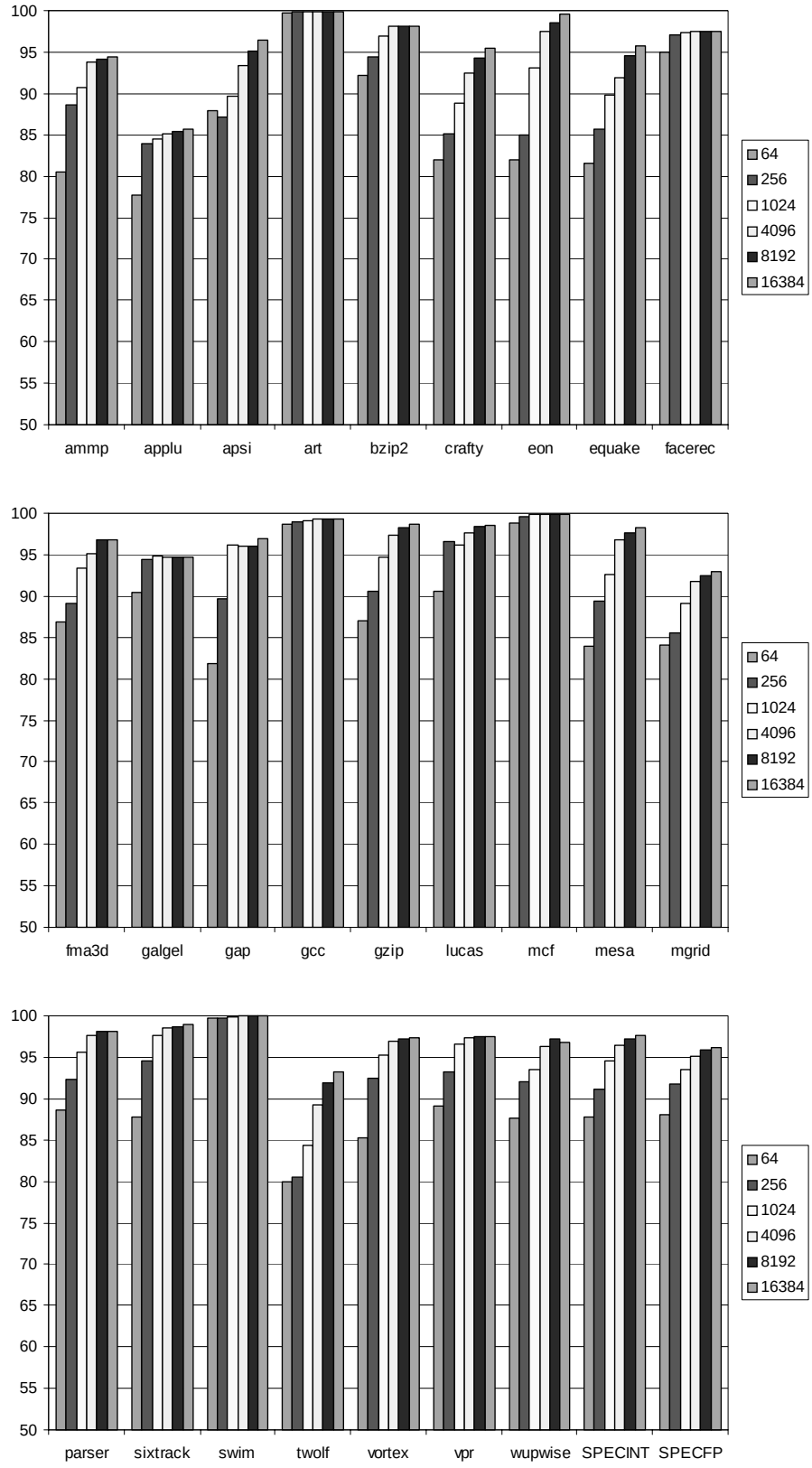


Figure 10. Prediction Accuracy of Last Tag Predictors of Various Sizes

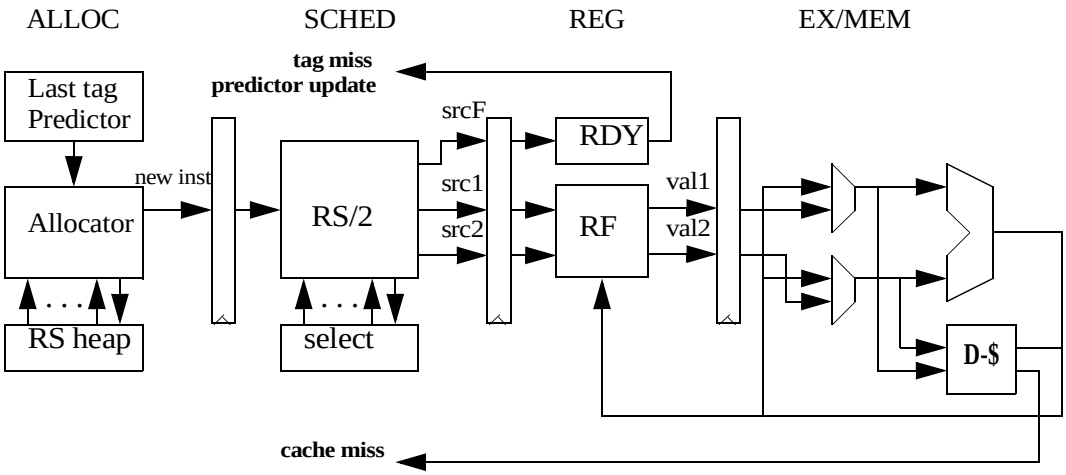


Figure 11. Scheduler Pipeline with Last Tag Speculation

A confidence estimation technique was also implemented for the predictor, but the results showed very few gains.

As shown in the pipeline of Figure 11, the allocator accesses the last-tag predictor for instructions with multiple unavailable operands and inserts the instruction into a reservation station with a single tag comparator. The predictor will indicate whether the left or right operand for the instruction will complete last. If the last tag prediction is correct, the instruction will wake up at the exact same time it would have in a window without speculation. In the event that the prediction is incorrect, the instruction will wake up before all of its input operands are ready, and a mispeculation recovery sequence will have to be initiated.

Figure 12 illustrates the datapaths and control logic for a reservation station supporting last-tag speculation. The input operand tags are loaded into the reservation station with the tag predicted to arrive last placed under the comparator (*srcL*). The other input operand tag (*srcF*) and the result tag are also loaded into the reservation station. The reservation station operates in a manner similar to a conventional design. Instructions request execu-

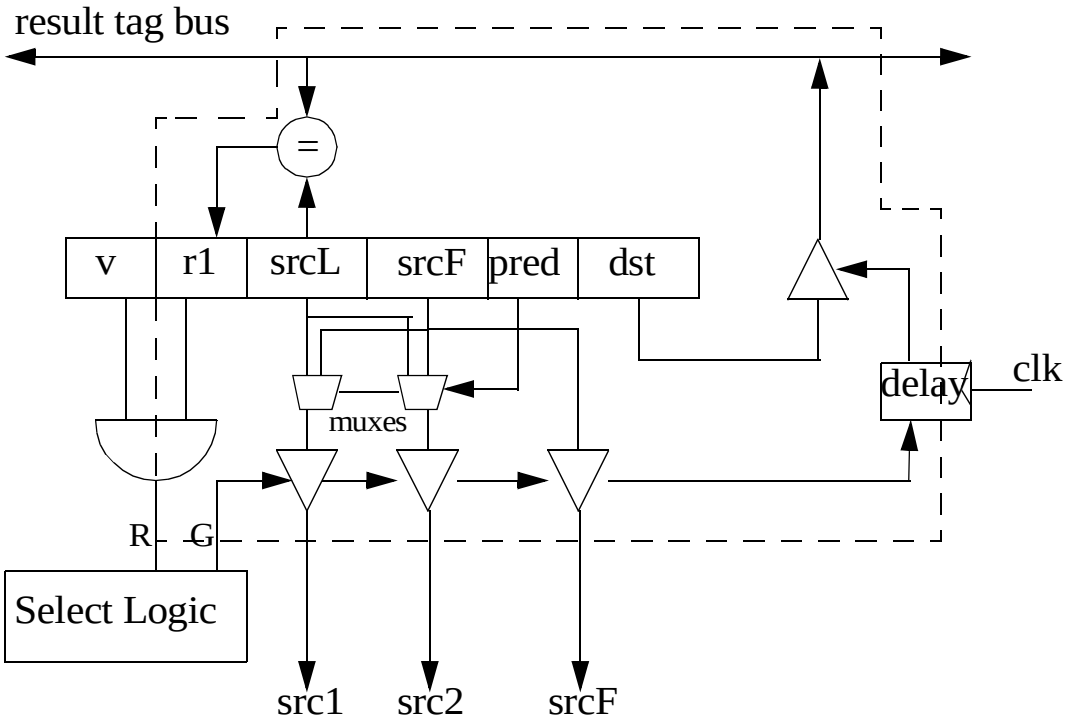


Figure 12. Reduced-Tag Reservation Station with Last Tag Speculation

tion once the predicted-last tag is matched on the result tag bus. When an instruction is granted permission to execute, the source operand register tags are driven out to the register stage of the pipeline. This drive operation requires a pair of muxes to sort the source operands into the original (left, right) instruction order, which is the format used by the register file and later functional units. In addition, the tag predicted to arrive first is forwarded to the register read stage (REG), where it is used to check the correctness of the last-tag prediction.

The last-tag prediction must be validated to ensure that the instruction does not commence execution before all of its operands are available. The prediction is valid if the operand predicted to arrive first (*srcF*) is available when the instruction enters the register read stage (REG) of the pipeline. In parallel with the register file access, the *srcF* tag is used to probe a small register scoreboard (*RDY*). The scoreboard contains one bit per

physical register; bits are set if the register value is valid in the physical register file. This scoreboard is already available in the ALLOC stage of the pipeline, where it is used to determine if the valid bit should be set when operand tags are written into reservation stations. An additional port to this scoreboard will suffice for validating last-tag predictions. Alternatively, an additional scoreboard could be maintained specifically for last-tag prediction validation.

If the prediction is found to be correct, instructions may continue through the scheduler pipeline as the scheduler has made the correct scheduling decision. If the prediction is incorrect, the scheduler pipeline must be flushed and restarted, in a fashion identical to latency mispredictions. Unlike latency mispredictions, which have a three cycle penalty, last-tag mispeculations only cause a one cycle bubble in the scheduler pipeline.

The primary advantage of the last tag scheduler is that more than half of the comparator load on the result tag bus is eliminated, which can result in reduced scheduling latency and significant power reductions for large instruction windows. The drawback of this approach is, of course, the performance impacts that result when a last-tag prediction is incorrect. Fortunately, the accuracy of the last tag predictor, combined with the small penalty for mispredicting should make this approach an effective technique for improving scheduler speed and energy consumption.

4.1.3 Selective Replay

In Kim and Lipasti's paper "Half-Price Architecture" [38], the authors introduce a clever selective replay implementation that performs parent-child dependence propagation using the scheduler broadcast busses. In addition, they correctly point out that the combination of this tag elimination scheme with broadcast-based selective instruction replay is

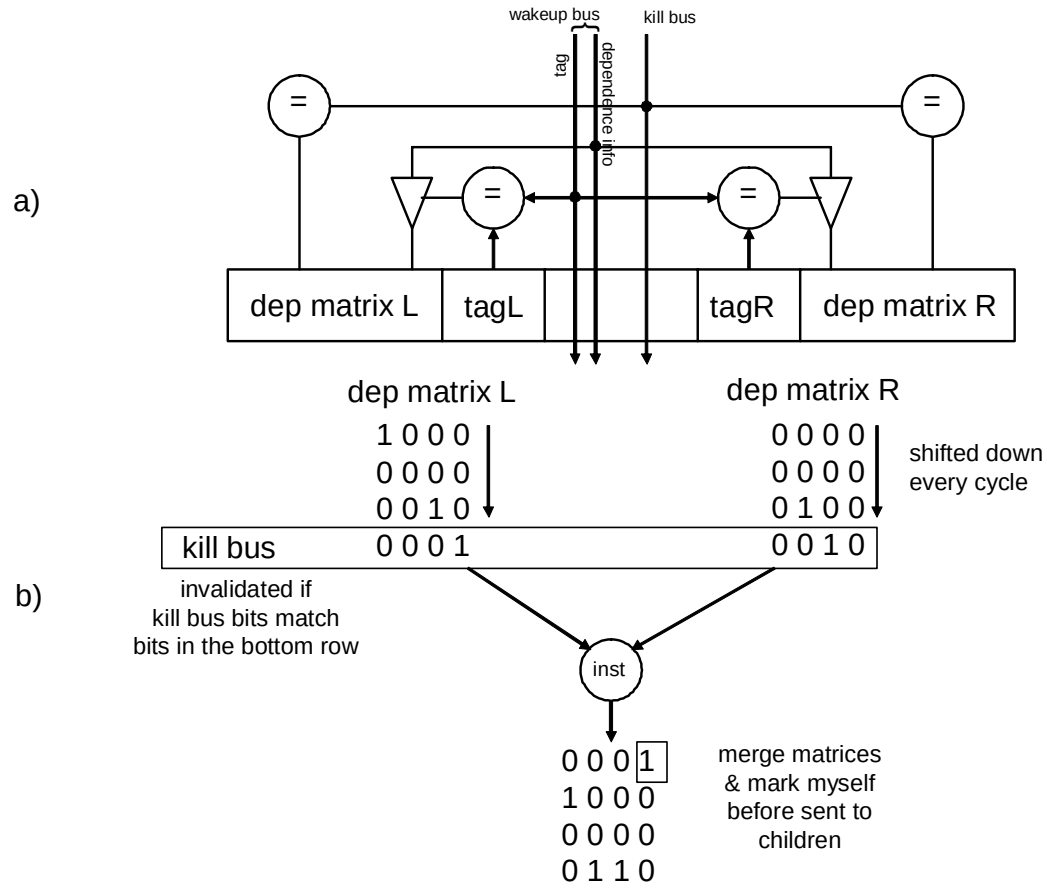


Figure 13. Half-Price Selective Replay Mechanism. (Figure from [38])

not practical to implement. In this section, we describe selective replay methods that are compatible with tag elimination. Furthermore, we analyze the performance and power consequences of the replay implementation decision.

4.1.3.1 Parent-Child Broadcast

In the Half-Price Architecture paper [38], Kim and Lipasti present one possible implementation of selective replay. An illustration of this scheme is shown in Figure 13.

Along with its input tags, an instruction's dependence information is kept in the instruction window in the form of one dependence matrix for each input operand. This

matrix consists of *W* x *D* bits, where *W* is the machine width, and *D* is the depth of the *load shadow* [9], which is defined as the number of stages between instruction issue and notification of a cache hit or miss. In each matrix position, the presence of a “1” indicates that an instruction is in the corresponding slot in the scheduler pipeline that the current instruction is dependent on, either directly or through some intermediate instructions. Every cycle, the matrix shifts down one row, to keep the information consistent with the movement of instructions through the pipeline ahead.

When a load latency misprediction occurs, the execute stage sets the appropriate bit in a *W*-bit wide array called the *kill bus*. These bits are broadcast to every instruction in the scheduler. As is illustrated in part b) of Figure 13, if the kill bus has a 1 in the same column as a 1 in the bottom row of a tag’s dependence matrix, that operand’s ready bit is reset to zero, as it is dependent on the instruction with the mispredicted latency. In other words, the matching bits indicate that the operand is dependent, either directly or indirectly, on the mis-scheduled instruction.

When an instruction leaves the scheduler window, it merges the dependence matrices that its input operands have received from their parent instructions and marks its own location. It then broadcasts this matrix, along with its destination tag, to the rest of the instructions in the window. (It must also write these bits into a table in the register rename stage for the benefit of dependent instructions that may have not entered the window yet.) When instructions in the window match the input operand on the tag bus, they also latch the dependence matrix of the broadcasting parent instruction. This process propagates dependence information from parent to child during the wakeup phase, giving them knowledge of ancestor instructions further up the dependence tree.

Tag Elimination Compatibility. As is discussed in [38], this selective replay scheme is not compatible with reduced-tag schedulers. Because reduced-tag scheduling makes decisions based on operand availability, problems arise when this availability information is allowed to change after instructions enter the window. Broadcast-based replay relies on every operand in the window tracking its dependencies. Because reduced-tag schedulers gain their complexity benefit by removing some operands from the tag bus, this is not possible.

In schedulers that use non-speculative tag elimination (*i.e.* they do not use last-tag speculation), instructions entering the window would get the proper dependence matrices from the table in the rename stage and they would still be able to monitor the kill bus. However, if the ready bit of an operand that has no comparator needed to be reset, there would be no way for that operand to return to snooping the tag bus.

Furthermore, removing the early arriving operands from the tag bus in last-tag speculation windows makes it impossible for those operands to receive their propagated dependence information from a broadcasting parent instruction.

4.1.3.2 Replay using Timed Queues

There are, however, other ways to implement selective replay. Some of these mechanisms differ from the parent/child broadcast model in that, instead of re-executing dependent instructions, they insert a delay into an instruction's execution latency, often through use of a queue or other type of separate instruction storage. The specific mechanism we outline here is derived from a technique proposed in U.S. Patent 6,212,626 [47], held by Intel for inventors Merchant and Sager of the Pentium 4 architecture team [28].[1] A block diagram of this design is shown in Figure 14.

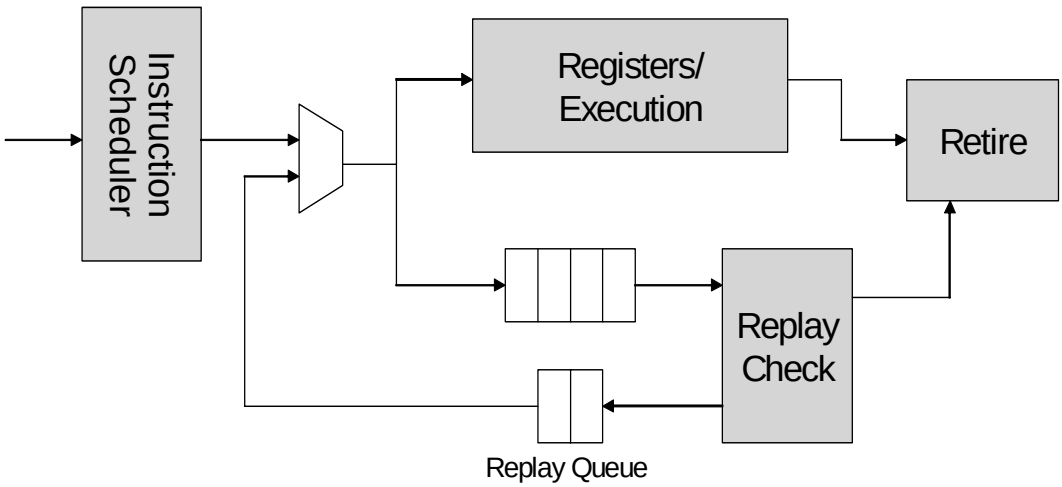


Figure 14. Intel Selective Replay Mechanism.

As instructions approach execution, they are also processed through a replay check, which consists primarily of a table of register ready bits. Just before entering the execute stage, instructions look up the status of their input operands in the checker. If the table indicates that the operands are ready, the instruction is allowed to enter execution and retire normally.

If, however, the check table indicates that an operand is unavailable, the instruction sets its output operand as not ready in the table and returns to execution via a replay queue and mux. The replay mechanism informs the scheduler of the presence of an approaching replayed instruction, so that nothing is scheduled into that slot in the same cycle. It is important to note that, in order to maintain forward progress, replaying instructions must always have priority over any work that would be coming out of the instruction scheduler.

1. Although we may refer to this as the “Intel” replay mechanism throughout this work, we are making no claim as to whether or not this technique is used in any of their microprocessors, commercial or otherwise. We are only presenting the idea proposed in the publicly available patent documentation.

When the replayed instruction reaches the input mux, it is sent back into the pipeline as if it had just been issued by the scheduler. On reaching execute, it checks its operands again, just as it did before, to determine whether it needs to replay again (An instruction may have to replay several times to tolerate an L2 cache miss, for example).

In this scheme, the propagation of dependence information is accomplished by the cascading ready bit manipulations in the check table. If there is a latency mis-prediction, the offending instruction's output will not be set as ready, which will trigger a replay for its children, which in turn will cause a replay for its children's dependents.

It is not specified in the patent exactly how many issue slots coming from the scheduler are stopped when an instruction replays. In our evaluation, we only prohibit the scheduler from issuing into the specific slot that the replaying instruction will be using. This allows the scheduler to issue instructions in the other issue slots.

Tag Elimination Compatibility. A key feature of the Intel replay mechanism is that it maintains the relative timings of instructions throughout the replay sequence. Once a mis-speculated instruction completes, its dependents are replayed just as they were originally scheduled out of the window, only the entire stream has been delayed to accommodate the unexpected extra latency. Consequently, there is no need to "re-schedule" instructions individually, as the previously selected schedule is still valid.

Because replayed instructions are not returned to the scheduler window, there is no extraneous dependence information kept in the window itself. Therefore, reduced-tag schedulers are fully compatible with the Intel-style selective replay.

In schedulers that use last-tag speculation, a last-tag misprediction still results in a one-cycle flush. This recovery is necessary to stop the wakeup of instructions dependent on the last tag misprediction.

4.1.4 Experimental Evaluation

The architectural simulators for this evaluation were derived from the SimpleScalar Toolset [6]. The circuit delays and power consumption statistics for scheduler windows used in this study were derived from an updated version of the SPICE models used in the work by Palacharla, Jouppi, and Smith [60]. All timing results are for the TSMC 0.18 μm process. A more detailed description of our architectural and circuit models can be found in Chapter 3. In addition, we estimated the power consumed by the last tag prediction array using CACTI II [62]. Table 1 shows the benchmarks, their instructions per cycle (IPC) on the baseline microarchitectural model, and their maximum baseline scheduler performance in instructions per ns (IPns).

The last-tag predictor configuration simulated is a GSHARE-style predictor with an 8-bit global history and a 8192 entry pattern history table. The global history is updated when branch instructions complete in the same way that the branch predictor is updated.

4.1.5 Performance of Reduced-Tag Schedulers

When reduced-tag reservation stations are introduced, instructions have a new constraint on entering the instruction window. Not only must there be an empty reservation station, but the station must also have at least one tag comparator for each of the instruction's unavailable input operands. If the demand for any particular class of reservation sta-

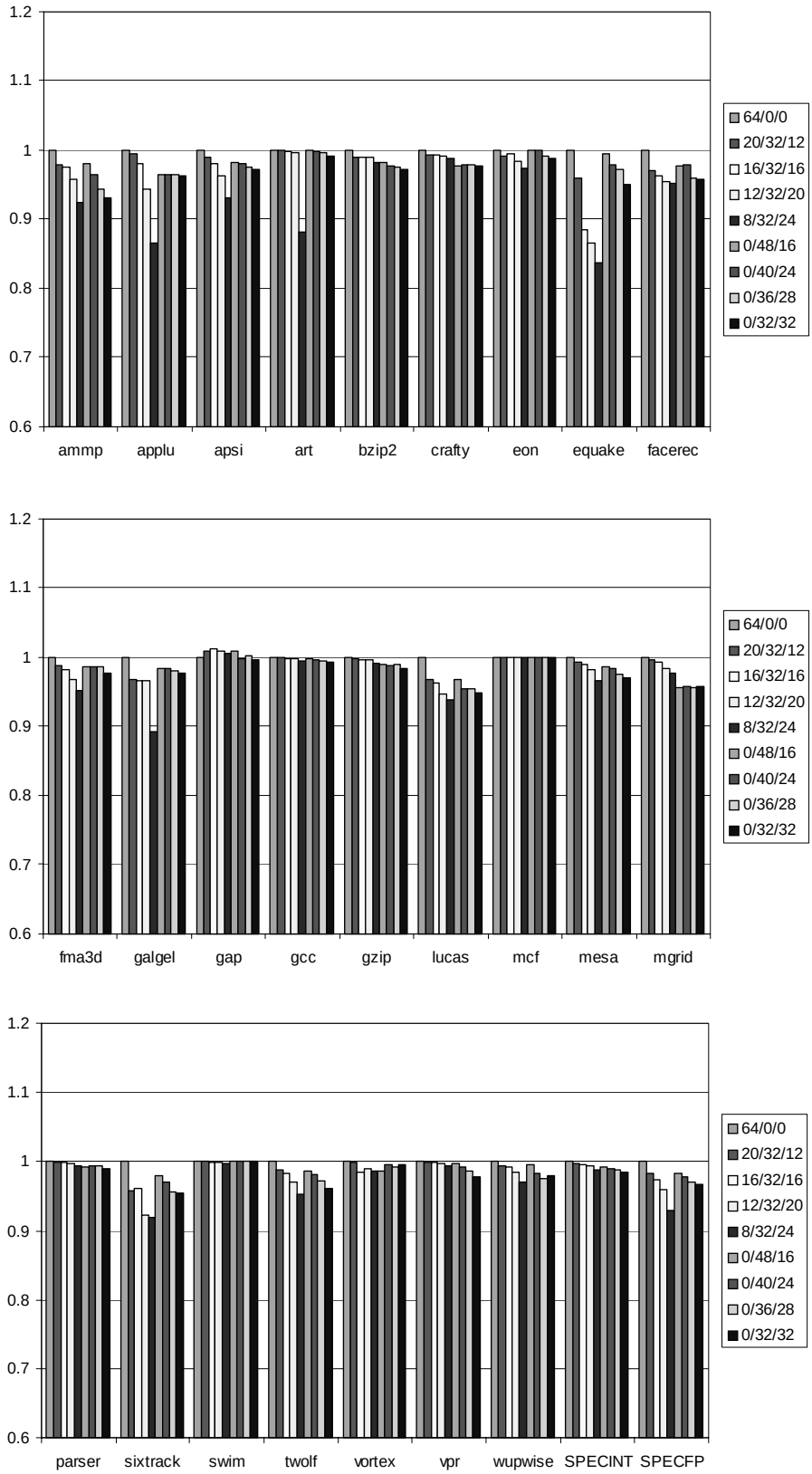


Figure 15. Instructions Per Cycle for Varying Configurations

tions is high, the reduced-tag designs may experience extra instruction stalls as the allocator waits for reservation stations to be freed.

These extra stalls reduce the effective number of reservation stations from which the scheduler can choose instructions to execute. The result, as shown in Figure 15, is a small IPC change for most configurations and benchmarks. We label the configurations “x/y/z”, where x, y, and z indicate the number of two, one and zero tag stations, respectively. Only the configurations without two-tag stations employ last-tag speculation. The effects of the stalls show up most prominently in *equake*. This benchmark makes extremely efficient use of the machine because it has excellent predictor performance, very few stalls, and many tightly coupled dependent instructions. Consequently, many instructions require the full two tag comparators. The configurations using last-tag speculation perform very well, with slightly lower IPC’s seen in benchmarks with poor branch predictor accuracy, such as *crafty.* In these programs, complex program control causes the register dependencies between instructions to change rapidly, making it more difficult to predict which operand will arrive last. The configurations without last-tag speculation slightly outperformed the configurations with speculation, with the exception of the 8/32/24 case, where there are a large number of stalls to wait on the small number of two tag entries. Overall, the performance impacts amounted to only 1-5%.

Figure 16 and Table 2 illustrate the circuit delay, power, and energy consumption for all analyzed scheduler configurations. In addition, the column in the table labeled $f_{tagload}$ lists the relative tag broadcast bus capacitive load, compared to the same-sized two-tag baseline design. This value indicates the relative decrease in comparator diffusion capacitance, and also reflects the relative reduction in tag bus wire length due to elimination of

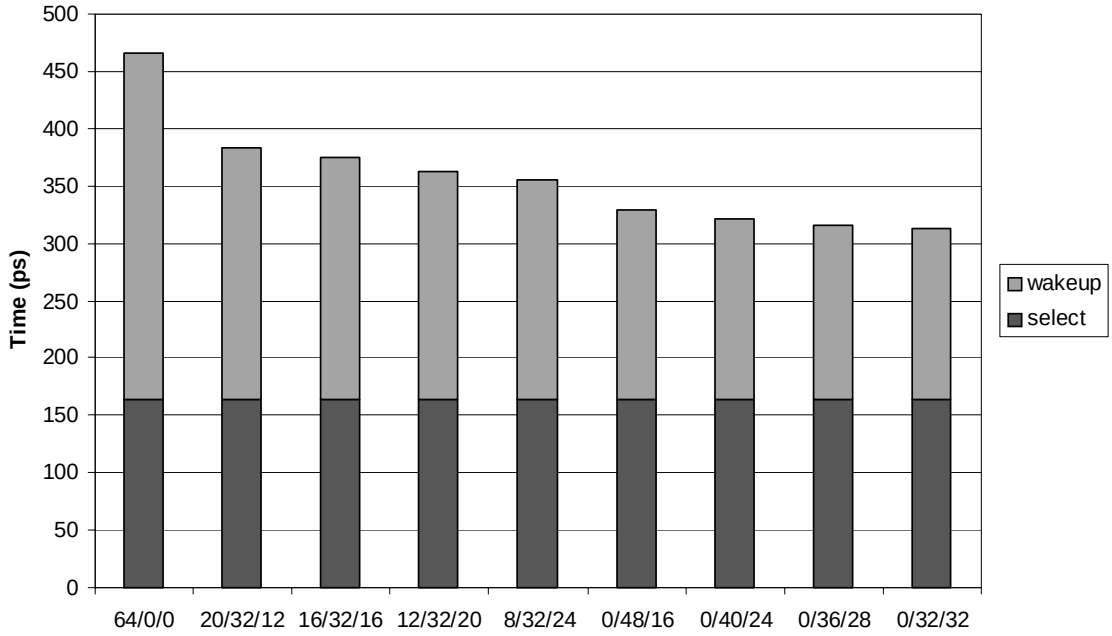


Figure 16. Scheduling delays for various tag elimination configurations

Table 2. Circuit Characteristics of Studied Scheduler Configurations.

Configuration	Total Delay (ps) (wakeup + select)	Total Power (W)	Total Energy (nJ)	$f_{tagload}$
64/0/0	466 (302 + 164)	1.004	0.468	1.0000
20/32/12	383 (219 + 164)	0.820	0.314	0.5625
16/32/16	374 (210 + 164)	0.773	0.289	0.5000
12/32/20	363 (199 + 164)	0.725	0.263	0.4375
8/32/24	355 (191 + 164)	0.673	0.239	0.3750
0/48/16	329 (165 + 164)	0.775	0.255	0.3750
0/40/24	321 (157 + 164)	0.692	0.222	0.3125
0/36/28	315 (151 + 164)	0.657	0.207	0.2813
0/32/32	313 (149 + 164)	0.610	0.191	0.2500
128/0/0	775 (573 + 202)	1.421	1.101	1.0000
40/64/24	552 (350 + 202)	1.308	0.722	0.5625
0/96/32	430 (228 + 202)	1.279	0.550	0.3750
32/0/0	349 (198 + 151)	0.605	0.211	1.0000
10/16/6	317 (166 + 151)	0.492	0.156	0.5625
0/24/8	290 (139 + 151)	0.466	0.135	0.3750

0-tag reservation stations and denser layout provided by the smaller 1-tag reservation stations.

The main benefit of removing tags from the scheduler critical path is the reduction in the load capacitance during instruction wakeup. Lower load capacitance allows for more aggressive clocking of scheduler circuitry. Based on our model of the wakeup and select circuitry, the specialized windows should allow for 25-40% faster clock rates, depending on configuration. Figure 17 shows the total performance (measured in instructions per ns for the scheduling stage) of each benchmark. With the exception of *equake*, the rate at which the scheduler can send instructions to execute measures between 20-50% higher, again depending on configuration.

4.1.5.1 Impact of Window Size

Figure 18 shows that reduced-tag scheduler optimizations continue to pay dividends for differing window sizes. The results given are the averages across all benchmarks. The gains become more prominent as total window size grows. The larger windows have fewer allocator stalls due to more reservation station resources. The large windows also bear more of the scheduler latency in result tag broadcasts (as opposed to the select logic), as a result, they show a larger percentage gain when tag comparators are eliminated. For a window with 128 entries, the optimized schedulers were 35-75% faster.

4.1.5.2 Energy and Power Characteristics

Often it is the case that to reduce power consumption, design changes must be made at the cost of lower performance. In our reduced-tag scheduler designs, lower load capacitance on the result tag bus provides both performance and power benefits. Table 2 shows

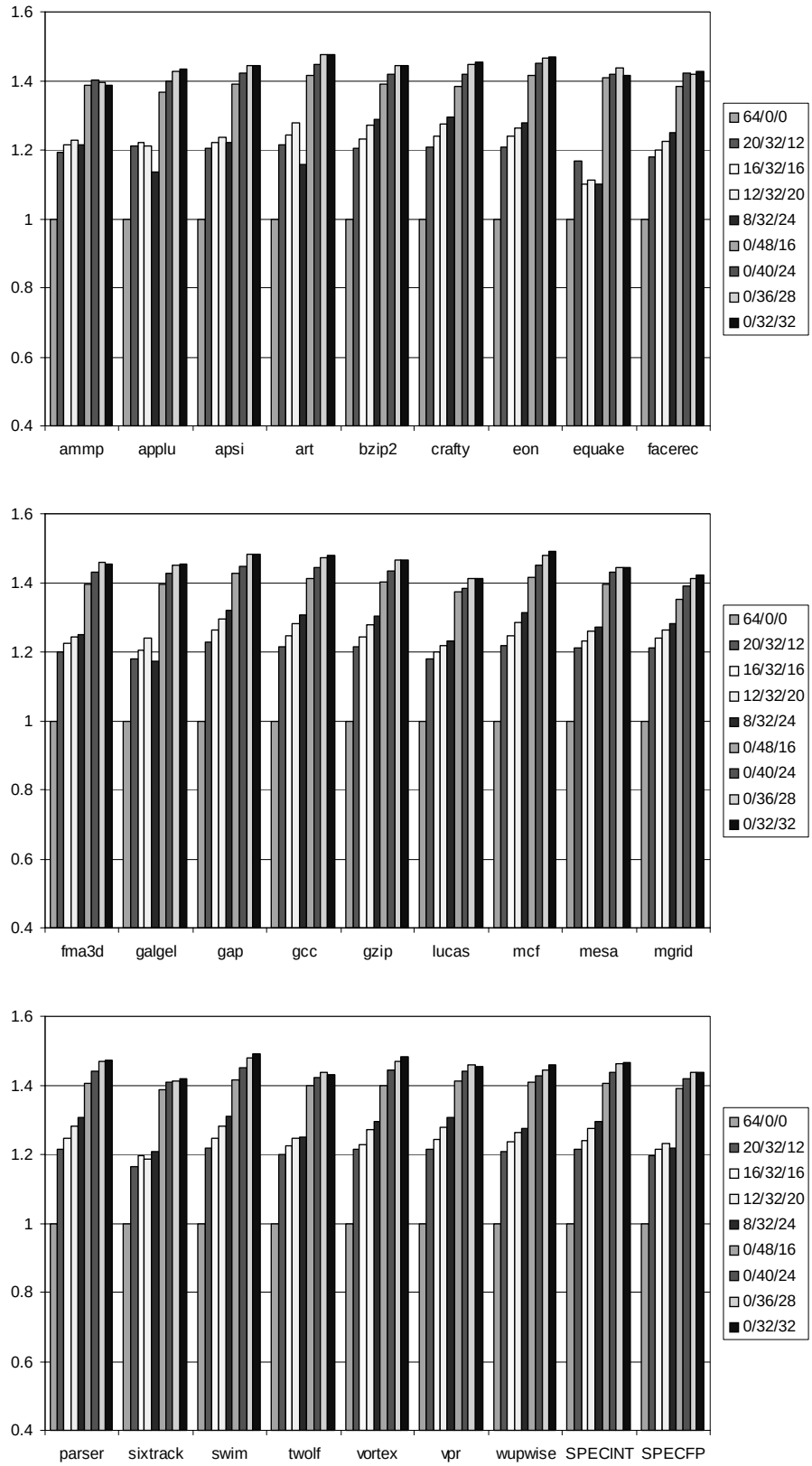


Figure 17. Instructions Per ns for Varying Configurations

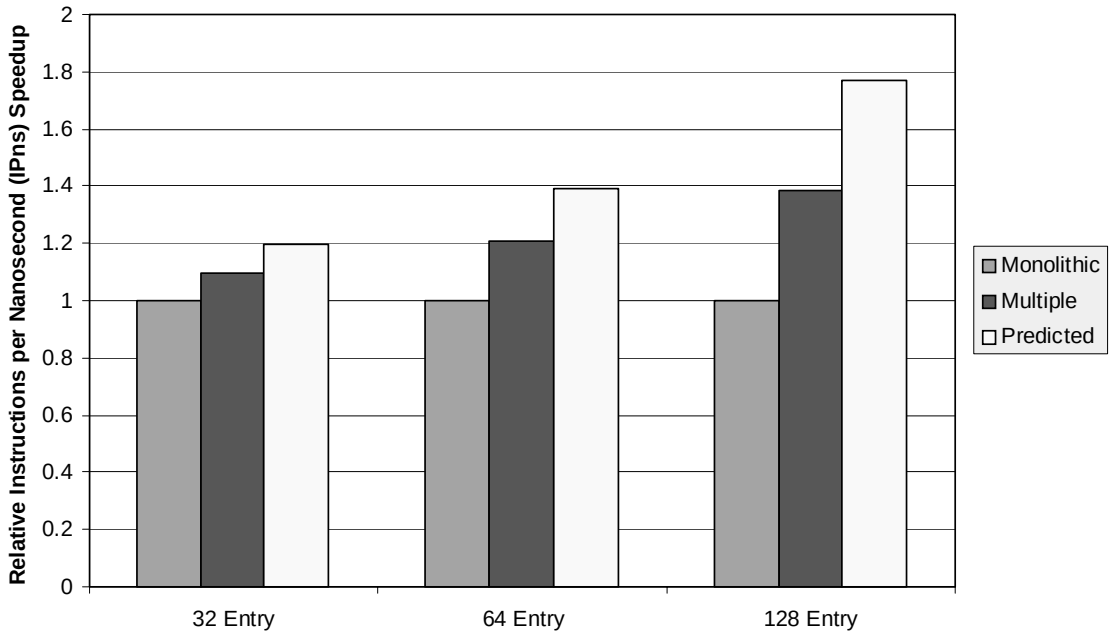


Figure 18. Impact of Tag-Reduction for Varying Window Sizes

the energy consumption for each configuration. The optimized designs use 30-60% less energy than the standard monolithic scheduler. Table 2 also shows the power used by each configuration if it were to be run at its maximum possible clock speed. The power reductions are not as pronounced as the energy improvements because the optimized designs run at a faster clock rate.

The power usage of the last-tag predictor was also calculated. It was found to consume less than 10% of the power used by the scheduler in all cases.

One way to quantify an architecture's ability to balance both power and performance is through the use of the energy-delay product [23]. This metric is the product of program run-time and total energy consumed to run the program. Figure 19 shows that the energy-delay product of the optimized scheduler is 50-75% lower than the baseline configurations. The 0/32/32 speculative configuration had the best return, with a 65-75% lower energy-delay across all experiments, including *equake*, which had the largest IPC impacts.

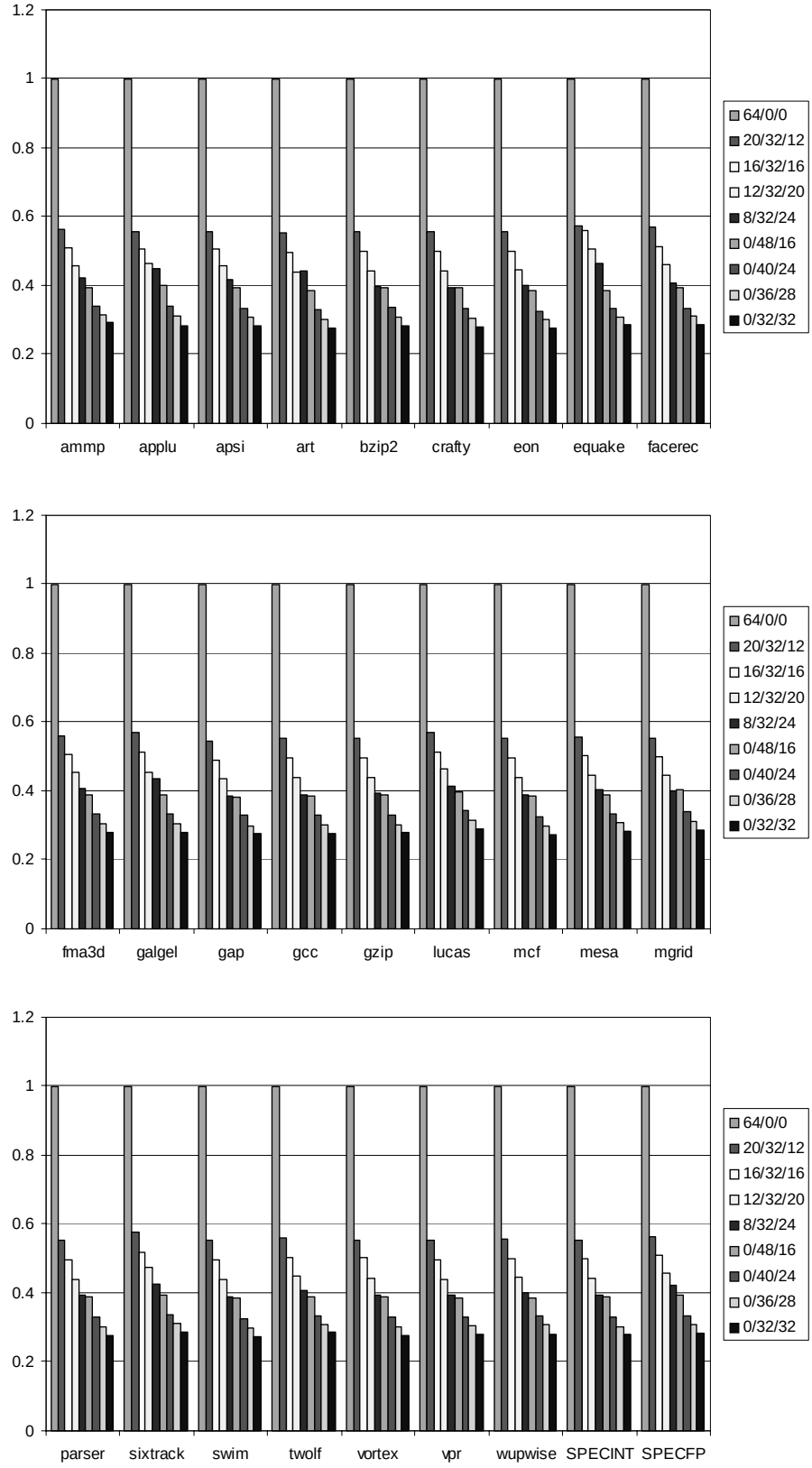


Figure 19. Energy-Delay Product for Varying Configurations

The optimized designs show large gains because eliminating tag comparators and tag bus wiring lowers the result tag bus capacitance, which both reduces energy consumption and allows for higher clock speeds. The energy-delay products for 128 entry windows also showed a 70% gain for the optimized configurations, suggesting that these benefits continue with larger window sizes.

4.1.5.3 Replay Evaluation

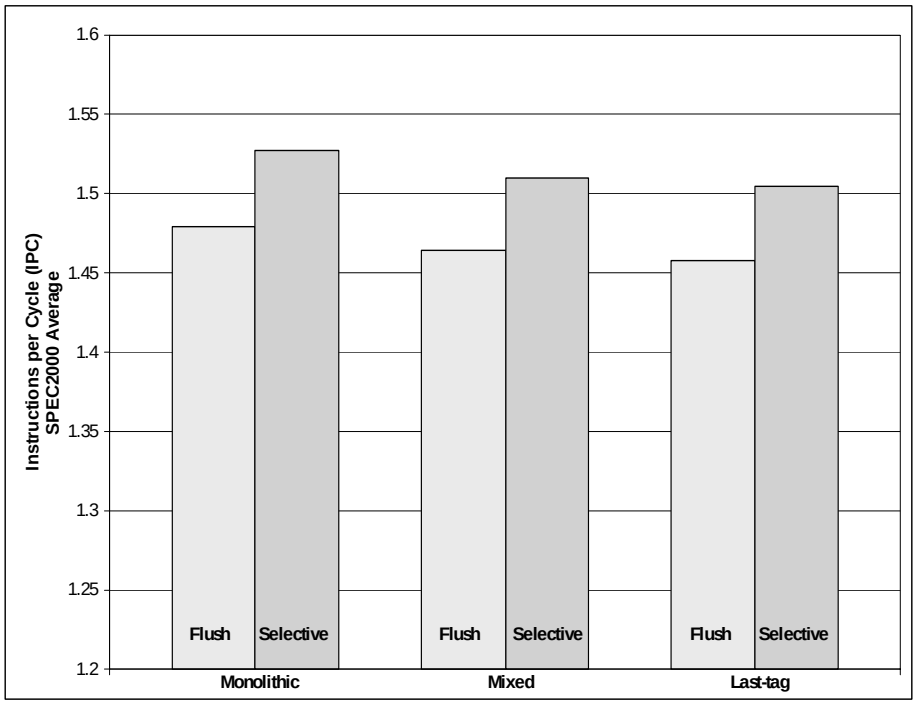
The baseline simulators discussed in Chapter 3 were modified to simulate either 21264-style flush replay [15] or selective replay using one of the two methods outlined in Section 4.1.3.

4.1.5.4 Tag Elimination and Replay

The SPEC benchmarks were simulated with three different schedulers on both 4- and 8-wide issue configurations, with the results shown in Figure 20. The baseline scheduler ("Monolithic") and the reduced-tag schedulers ("Mixed" and "Last-tag") all gain 2-3% performance improvement due to the decreased replay penalty. *Galgel* benefitted the most with a 26% improvement due to a large number of memory references and enough parallelism to suffer from pipeline flushes. No benchmarks saw a performance degradation due to selective replay. The performance improvement of 2-3% would close much of the gap demonstrated in the experiments of Kim and Lipasti [38].

4.1.5.5 Instruction Window Pressure

As is discussed briefly in Borch *et al.*'s work [9], the parent-child broadcast replay model requires that instructions must remain in the scheduler window for several cycles after they are issued, in order to monitor the kill bus for a replay indication. When all of an



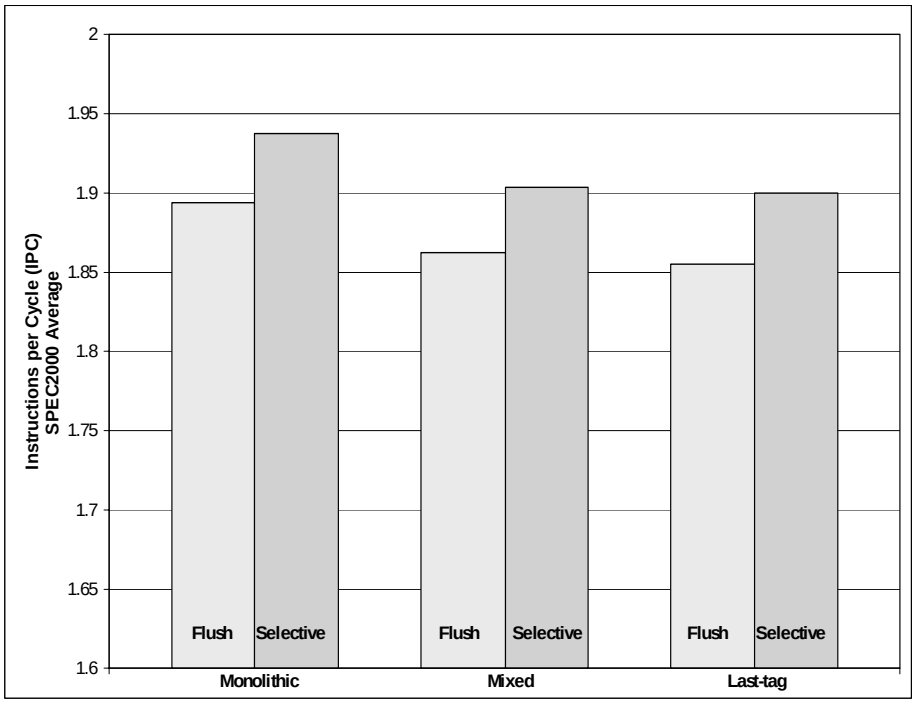


Figure 20. Effect of Selective Replay on Reduced-Tag Schedulers.

instruction's ancestors are safely into execution, only then will it finally release its reservation station.

As a result of keeping instructions in the window beyond their issue time, this mechanism can suffer from a reduction in effective scheduler window size. For example, an 8-wide machine with a 4-cycle *load shadow* could be holding as many as 32 instructions in the scheduler window that have already issued, reducing the number of spots that are available for newer instructions. For the typical case, the number of extra instructions held in the window will not be that large because the processor will not usually be filling all of its issue slots.

Using the Intel replay technique, instructions never re-execute out of the scheduler window, thus removing the need to stockpile instructions after they've issued. This reduces the instruction pressure in the window, allowing more work to flow into the empty slots.

On the other hand, the Intel approach can limit execution bandwidth when too many instructions are in replay, thus preventing new instructions from entering execution. However, if there is a large number of replaying instructions, either they are all waiting for one long-latency load, or there are multiple outstanding latency mispeculations. In either case, it is not likely that much more parallelism could be found anyway.

Schedulers with 32, 64, and 128 entries were simulated using both replay techniques, with the results shown in Figure 21. As intuition would suggest, the most benefit was seen in configurations with smaller windows and wider issue, with the 32-entry 8-wide scheduler receiving a 5% performance improvement from the reduced instruction window pressure.

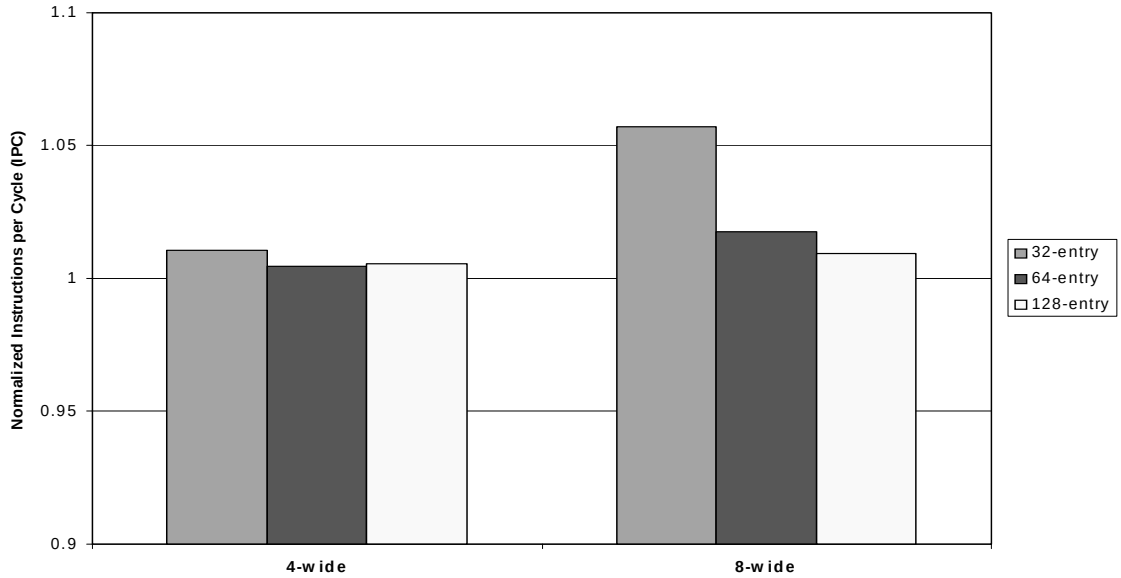


Figure 21. Effect of Reduced Scheduler Pressure. Results presented show the relative performance of a scheduler using the Intel replay mechanism with respect to a scheduler of the same size with a broadcast-based mechanism.

4.1.5.6 A Discussion on Replay Power Consumption

In the parent-child broadcast replay mechanism, the dependence matrix of an issuing operation is sent to all other instructions in the scheduler window. As has been shown, these wire-intensive broadcasts can be very costly from a power standpoint.

The load on the broadcast bus as seen by each dependence matrix bit can be estimated as roughly equivalent to that seen by each destination tag bit. While each bit of the matrix bus could be separated from the matrix latches by a low-capacitance pass gate, the bus line must still be able to drive the input of the latch if this gate is open. While having all of the pass gates open is an extreme case (all instructions depend on the broadcast through both operands), it is necessary to take it into account as a peak case. The kill bus bits will also

have the same load as the tag bits, since they are also being driven to comparators for each operand.

In a standard instruction scheduler without this replay mechanism, the number of bits broadcast each cycle is

$$W \times (\text{dest tag bits}),$$

where W is the scheduler issue width and the number of destination tag bits is $\log_2(\#\ \text{of physical regs})$. If the parent-child broadcast mechanism is incorporated, the number of bits broadcast each cycle is

$$W \times (\text{dest tag bits}) + (W \times (W \times D)) + W,$$

where D is the number of cycles in the load shadow. The first portion of the equation represents the destination tag broadcasts, and it is the same as for the standard window. The second and third terms of the equation represent the dependence matrix bits and the kill bus bits, respectively.

This drastic increase in broadcasts may not directly alter the cycle time (although the layout expansion could have some effect). However, the power consumption will likely be noticeably larger. For example, an 8-wide window with a load shadow of 4 and 256 registers will need to broadcast 328 bits across the scheduler instead of just 64.

The Intel replay technique requires none of these extra broadcasts. The mechanism does include some extra logic, but the power consumed by the check table should be far less than the amount that would be dissipated across wire-intensive broadcast lines. This comparison is similar in scope to the comparison of the power usage of a last-tag predictor table with the power used by the scheduler window as was discussed in the previous section.

4.1.6 Summary

The tag elimination approach we have presented greatly reduces the complexity per entry on a CAM-based dynamic scheduler. Designs using these techniques gain not only large amounts of breathing room under pipeline timing constraints, but benefit from significant power savings as well due to significantly lower bus capacitance.

While much better than a baseline design, tag elimination designs still suffer from the basic problems inherent in all CAM-based designs: poor performance and power scalability when window sizes are increased or when fabricated in newer process technologies. These issues are created by the basic broadcast nature of the structures, where instructions are required to communicate their results to all other instructions that might need them.

In the following section, we describe a different design, called Cyclone, which removes the broadcast element from dynamic scheduling, allowing us to break out of the poor scalability curve.

4.2 Cyclone

Many researchers have explored ways to mitigate the scalability problem with dynamic schedulers. Work by LeBeck [40] and Morancho [50] relieve pressure on the instruction window by removing long-waiting instructions. The efforts of Raasch [61], Goshima [25], Gonzalez [13][21], and many others, have attempted to remove as much unnecessary circuitry as possible from the critical path, in the same vein as the tag elimination work just described.

The principle drawback of all of these designs is that they still depend on the CAM broadcast mechanism to generate their final schedules. This exposes them, to a varying extent, to the same eventual scalability problems that this mechanism is plagued with.

In light of this, it became our goal to devise a way to generate a schedule and implement that schedule without using global broadcast circuitry. The first step in this endeavor was to explore non-broadcast methods for generating instruction schedules.

The most obvious alternative to standard dynamic schedule generation can be found in compiler-scheduled VLIW machines. As shown in Figure 22, it is possible to implement instruction scheduling at compilation or during execution. Compile-time instruction schedulers analyze the flow of control and data in a program, and then re-order instructions in the binary so that during execution they will better utilize processing resources. The primary advantage of compile-time instruction scheduling is that the hardware necessary to carry out the execution of instructions can be very simple. For example, in machines such as Itanium [34], the compiler selects “bundles” of independent instructions that are fetched and executed as a single unit, obviating the need for any complex dependence checking logic in the underlying hardware.

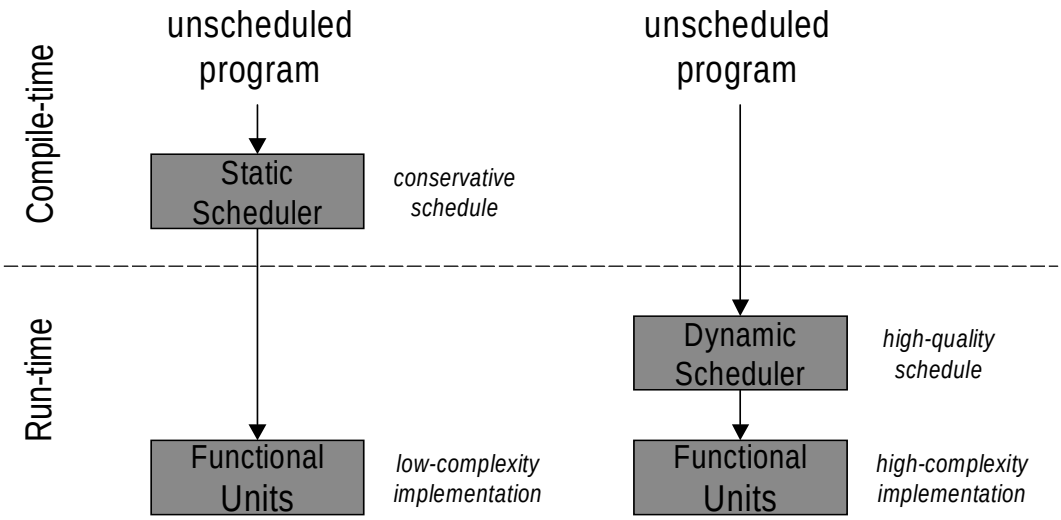


Figure 22. Compile-time vs. Run-time Instruction Scheduling.

The primary disadvantage of compile-time scheduling is that it lacks an accurate assessment of dependencies caused by branches and memory operations, as these are a function of program execution. Consequently, any scheduling decisions implemented in the presence of these instructions must be made conservatively. The conservative tendencies of compile-time schedulers are well recognized, and as such, many proposals have been made to lessen their effect. For example, branch boosting [68] and predication [27] mitigate the effects of control dependencies by pre-computing or hiding branch instructions. Advanced loads [34] and run-time disambiguation [57] have been proposed to reduce the impact of ambiguous load/store dependencies.

As has been shown, dynamic scheduling at run-time holds the opposite disposition of compile-time scheduling. Since program dependence analysis occurs at run-time, branch directions and load/store addresses are often available to improve the accuracy of scheduling. Modern designs go another step further and incorporate branch and load/store dependence predictions to further refine the schedules. However, the hardware necessary to

implement dynamic scheduling can be quite complex and slow, as has been discussed extensively.

In this section we present the *Cyclone* scheduler, a design that draws from compile-time and run-time scheduling techniques in an effort to secure the low-complexity of a compile-time scheduler and the high-quality instruction schedules of a dynamic scheduler. The hardware-based instruction scheduler implements a simple scheduling algorithm similar to compile-time list scheduling [52]. Instructions are scheduled based on the predicted latency until the availability of its operands. To obtain high-quality schedules, we employ our scheduling algorithm at run-time where it is implemented by a simple hardware unit in the front-end of the processor pipeline. To further improve schedules, the hardware-based list scheduler incorporates branch predictions and load/store dependence predictions to speculatively orchestrate execution in the presence of control and memory dependencies. Once an instruction's latency has been predicted, it enters the Cyclone scheduler queue, which implements a network of locally-synchronized datapaths that route an instruction to its functional unit at (or shortly after) its proscribed execution time. To ensure correct program execution in the presence of latency and dependence speculation, the Cyclone scheduler incorporates a selective replay mechanism that overloads the register forwarding infrastructure to re-execute only those instructions dependent on incorrectly scheduled instructions. This same mechanism is used to identify and flush instructions which have been squashed due to mispeculation.

The Cyclone scheduler makes three substantial contributions in the area of high-performance dynamic instruction scheduling. They are as follows:

- Broadcast free dynamic scheduling: The Cyclone scheduler contains no global broadcast or control signals. Our design employs distributed methods to generate an instruction schedule, re-synchronize impaired schedules, and recover from mispeculation. The lack of global control enables very fast clocking and uses much lower area due to reduced interconnect requirements.
- Efficient dependence-based variable-latency instruction replay: In the event of a mispredicted latency or dependence, the Cyclone scheduler is capable of recovering schedules by selectively replaying only those instructions dependent on the instruction forcing the replay. The latency of a replay may be variable, and it can be easily set by the instruction that initiated the replay.
- First-class scheduling of memory dependencies: Our design incorporates store-set predictions [14] into the scheduling mechanism and treats store/load dependencies in the same fashion as register dependencies. As a result, our simulations and timing analyses fully account for the impacts of memory communication on the throughput of the scheduler.

4.2.1 The Cyclone Scheduler

4.2.1.1 High-level Architecture

Figure 23 illustrates the high-level architecture of the Cyclone scheduler. After register renaming, instructions enter the *instruction pre-scheduler*, which predicts the number of cycles that must elapse before the instruction's operands are available for execution. Once an instruction's latency has been predicted, it enters at the tail of the *countdown queue*. In the countdown queue, instructions move over locally-synchronized datapaths

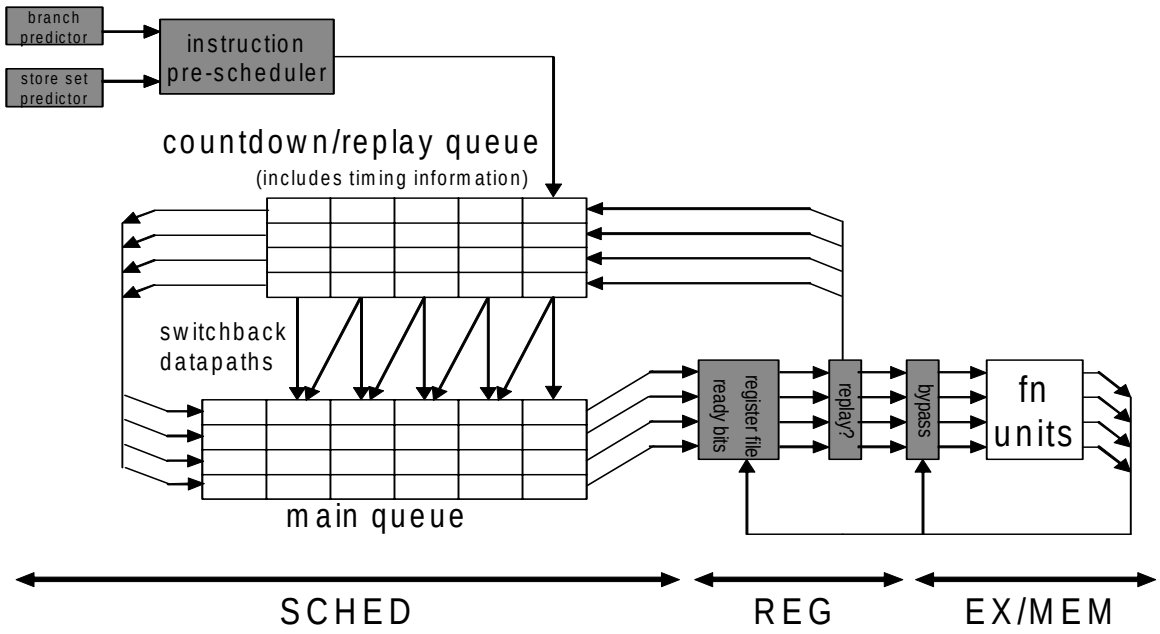


Figure 23. Cyclone Scheduler Architecture

toward the end (left side) of the queue at the rate of one queue entry per clock cycle. When one half of the predicted latency until execution has expired, instructions jump to the lower *main queue* via *switchback datapaths.* Like the countdown queue, the main queue steps instructions toward execute at the rate of one entry per clock cycle, permitting instructions to arrive at execute at (or near) their predicted execution time.

To facilitate the construction of high-quality instruction schedules, the Cyclone scheduler supports control and data speculation. Load/store dependencies, branch directions and load dependencies are all speculated by the instruction scheduler using a branch predictor and a store-set predictor [14]. If the speculative schedule becomes corrupt (due to, for instance, an input sourced by a load that missed in the data cache, or a late switchback), the Cyclone scheduler is capable of repairing the schedule on-the-fly using a selective replay mechanism that re-executes only those instructions dependent on incorrectly scheduled instructions.

To implement selective replay, instructions access physical storage ready bits immediately before attempting execution. If an instruction arrives at execute and its operands are ready, it commences execution as planned. If the instruction's operands are not yet ready, the latency until execution is once again predicted, and the instruction is re-inserted back into the countdown queue. In addition, the physical register destination of the replayed instruction is marked unavailable in the ready bit table. Subsequent instructions that are dependent on this operation will find this operand unavailable and replay as well. Using this dependence-based replay approach, only those instructions that use the result of an incorrectly scheduled instruction must be replayed. It is important to note that the replay mechanism is sufficiently robust that it can detect and correct any scheduling errors by replaying instructions until their operands arrive. As such, the instruction pre-scheduler need only act as a schedule predictor, any errors it introduces into the instruction schedule will be safely corrected by the replay mechanism.

Wide issue is supported by the Cyclone scheduler by providing additional capacity for scheduling and timing instructions within the Cyclone scheduler queue. It is possible to increase the capacity of the scheduler queues by providing space for multiple instructions in each entry. We term these additional instruction slots *rows*. The scheduler depicted in Figure 23 contains a 4-row Cyclone scheduler queue.

The Cyclone scheduler has lower circuit complexity than conventional broadcast-based scheduler designs (such as a Tomasulo [27] or matrix scheduler [25]) because it lacks centralized control mechanisms of any kind. All communication occurs locally between instruction entries in the Cyclone scheduler queue. Comparatively, broadcast-

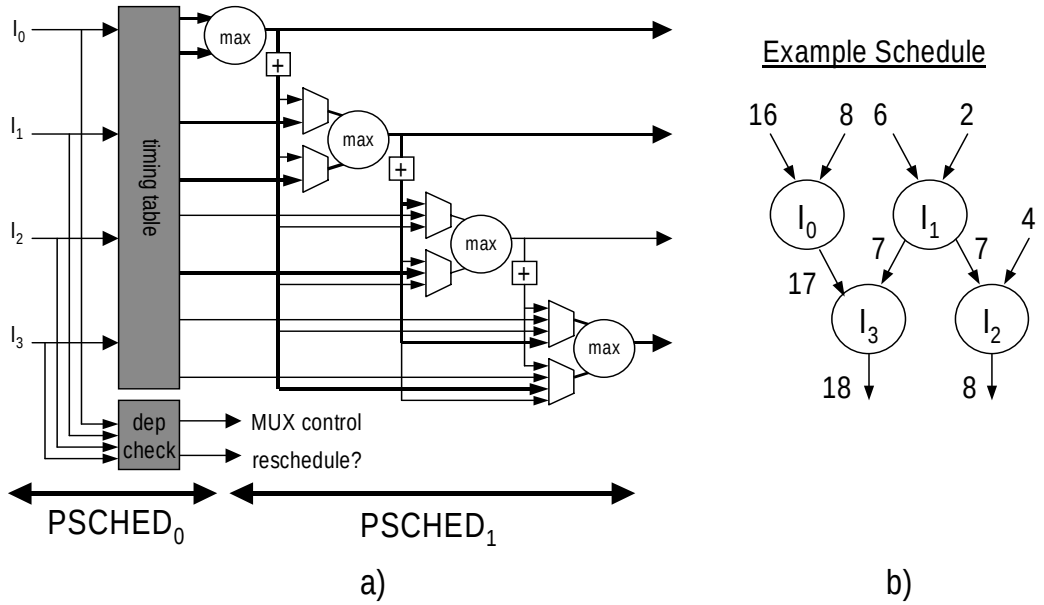


Figure 24. Pre-scheduler Design and Example. The front-end pre-scheduler architecture is shown in a) and an example chain of four dependent instruction (with delays until execute) are shown in b). All instruction operations in the example execute in a single cycle.

based schedulers must send dependence information to all other instructions each cycle, making highly capacitive broadcast wires with long propagation delays a necessity.

4.2.1.2 Instruction Pre-Scheduler

The instruction pre-scheduler, illustrated in Figure 24a, is responsible for scheduling instructions into the instruction queue such that instructions reach the execute stage immediately after their inputs are computed. Our design is a variant of the data flow prescheduling design presented by Gonazalez [21], and elaborated on by Michaud [48]. The instruction pre-scheduler resides in the front-end of the pipeline, after instruction renaming and before the execution queues. Each cycle, instruction placement is computed by the scheduler for each instruction in program order. Instructions are placed at the tail of the countdown queue such that, if an instruction is expected to execute in N cycles, it will be

given a timer value that will prompt it to cross over to the main queue in N/2 steps. For a W-wide decode width processor design, the instruction scheduler places the next W dynamic instructions into the tail of the countdown queue each cycle.

Conceptually, the instruction pre-scheduler computes, for each instruction, the delay until all inputs are available. This task is accomplished with a simple timing table and a MAX calculation. An instruction with two inputs available at times t_0 and t_1 can begin execution at MAX(t_0, t_1). The result of that instruction will be available at MAX(t_0, t_1)+L, where L is the latency of the instruction operation. As shown in Figure 24a, each instruction accesses the *timing table*, which produces the delay (in cycles) until the input is available. Each instruction then computes the maximum delay of its inputs, and the result is used as the index to place the instruction into the queue.

The instruction scheduler timing table is indexed with the logical register index, and it returns the delay until the operand is ready. For instructions with immediate operands or no operands (e.g., load-imm), the timing table returns a zero delay. When an instruction is scheduled, its schedule time is written into the scheduler timing table under the destination register index. To allow the scheduling of dependent instructions in back-to-back fetch groups, the destination delay values are also forwarded back into the pre-scheduler logic.

4.2.1.3 Limiting Dependence Chains

A complication arises in the design of the pre-scheduler because instructions in an issue group may share dependencies between each other. These dependencies create a recurrence in the scheduler timing computation such that the input to the MAX computations may be the result of the MAX computation of earlier instructions in the same group. We have performed extensive simulations with a range of designs for dealing with inter-

instruction dependency scheduling. One conclusion we reached fairly quickly was that it is insufficient to issue only the independent instructions in each cycle. This insight is not surprising in retrospect, as placing an in-order issue mechanism anywhere before the dynamic scheduler should greatly degrade the throughput of the dynamic schedule.

The scheduler design presented in Figure 24a allows dependent chains of instructions to compute their correct start times within the same cycle. The output of each MAX calculation can be forwarded to any later instruction, to override an input with an earlier computed schedule time. In the first cycle of schedule ($PSCHED_0$), the timing table is probed to determine the delay until input operands are available. In the second cycle of scheduling ($PSCHED_1$), the MAX function computes the time the instruction operands are available, and forwards this results to the inputs of all MAX computations in later cycles.

While the datapaths in the circuit would permit arbitrary length chains of dependent instructions (up to four in the figure), *the dependence logic limits the computation to at most two cascaded MAX computations*. Simulations indicate that dependence chains longer than two instructions happen in less than 3% of all fetch cycles for SPEC2000. Our MAX calculations are limited to 6 or 7 bit subtraction operations (depending on maximum Cyclone queue depth); consequently, the resulting scheduler is both fast and accurate. In the event an instruction chain length is longer than two, the dependence check logic will indicate this case by the end of the $PSCHED_1$ cycle by asserting the *reschedule signal*, which forces an additional cycle to complete the schedule times of long dependent chains. This additional cycle causes very little degradation in performance, since the instructions at the end of long chains would not be ready to execute immediately anyway.

An example of dataflow instruction scheduling is illustrated in Figure 24b. Instructions I_0 and I_1 compute their ready times and forward them to the inputs of instruction I_3's MAX calculation. Instruction I_1's MAX calculation is also forwarded to one input of instruction I_2. After two cascaded MAX calculations, all four instructions have correctly computed their schedule times. It is interesting to note that in a VLIW machine instructions within a fetch group are independent. This would further simplify the instruction pre-scheduler because MAX computations would never need to be forwarded to later MAX computations in the same fetch group.

To keep the pre-scheduler logic simple, it does not accept new instructions until the current fetch group has been fully scheduled. In addition, the instruction scheduler will stall when the queue entry for any of the scheduled instructions is full. In any of these events, the scheduler will retry on subsequent cycles until the full instruction fetch group is scheduled and inserted, at which point scheduling may continue.

4.2.1.4 Memory Scheduling

It is vital to the production of accurate schedules that loads be scheduled as soon as possible after dependent stores, otherwise, load/store dependencies will cause a significant number of instruction replays. These replays increase the pressure in the Cyclone queue, potentially preventing later instructions from entering the main Cyclone queue, and delaying the execution of loads for a complete trip through the replay loop. To accurately schedule loads, we have adapted the store set dependence predictor [14] to time the execution of load instructions. The store set predictor tracks the stores that are a source for a particular load and assigns to that group a store set identifier. Loads are scheduled after the last store from the store group still in flight when the load is dispatched. The approach can be

readily adapted to the Cyclone scheduler by recording with each store set identifier the delay until the last store in flight completes. When scheduling stores, the latency of a store operation is equal to the time to forward a store value to a load instruction, typically the latency for the load/store queue forwarding mechanism (one cycle in our experiments).

4.2.1.5 Switchback Datapaths

To implement the instruction schedule computed by the pre-scheduler, instructions are injected into the tail of the Cyclone scheduler queue with a prediction of how far the instruction should progress down the countdown queue before turning around and heading back toward execution in the main queue. Switchback datapaths provide the connections over which instructions can turn around and head back toward execution. Their inclusion in the design is vital to keeping the scheduler circuit complexity low. With them, it is possible to eliminate any random access ports into the scheduler queue. Instead, all instructions must enter at a single point into the tail entry of the countdown queue, after which they work their way to execution to meet their predicted delay.

Figure 25 illustrates the switchback datapaths and control logic. Instructions time their entry into the main queue using simple countdown logic. Instructions are injected into the countdown queue with a timer value equal to half of their total predicted latency. When the countdown completes the instruction will begin attempting to switchback to the main queue. As shown in Figure 25a, instructions with an even numbered latency jump down to the entry directly below (m0), while instructions with odd numbered latencies jump down and to the left (mp). This routing approach inserts an additional step between entry and execute for instructions with odd numbered predicted latencies. Without the diagonal data-

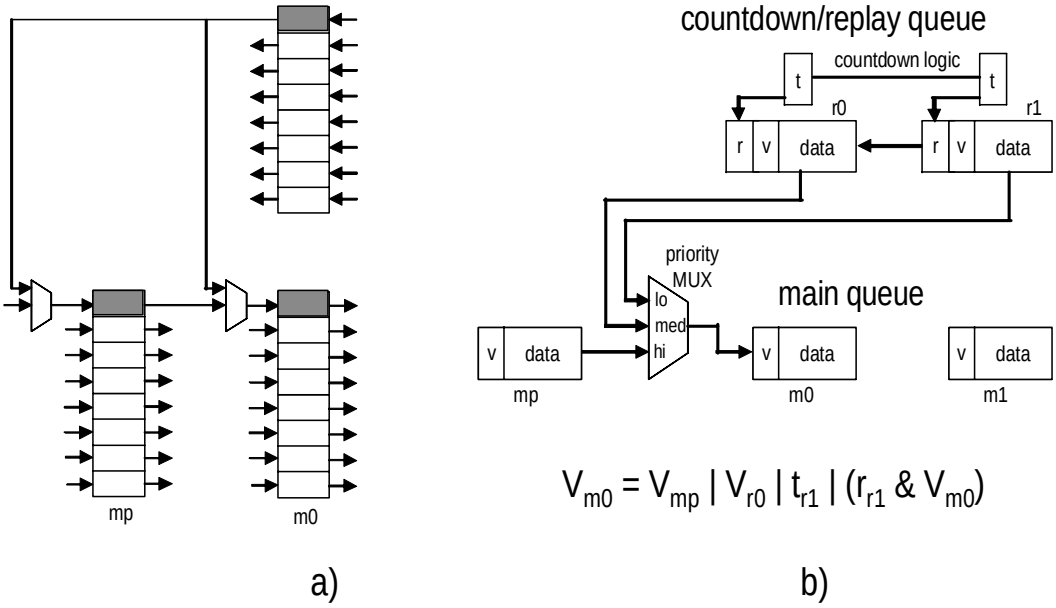


Figure 25. Switchback Logic. The options an instruction has for crossing from the countdown or replay state to the main queue are shown in a). A diagram of this logic for a single instruction, and its critical path logic equation is shown in b).

paths, all latencies would have to be even as the same number of queue entries would be traversed down and back to reach execution.

To keep countdown logic simple, we employ a cascaded Johnson counter [85], which requires N bits to count a maximum of 2N states. The advantage of the Johnson counter is that it only requires one inverter per entry in the countdown queue. An additional feature of the Johnson counter is that we are able to pick counter states such that the most-significant bit of the counter state transitions when switchback should occur, eliminating the need for any comparison logic on counter values.

Because main queue entries can accept instructions from multiple sources (e.g., mp, r1, and r0 in the figure), request conflicts must be resolved. In the event of multiple requests, highest priority is given to the previous main queue entry (mp). This entry must

have highest priority because it has nowhere else to go and it cannot stall. If the previous main queue entry is empty, priority is next given to the above switchback channel (r0), followed by the channel diagonally above and to the right (r1). If countdown queue entries cannot enter the main queue due to request conflicts, they will instead enter the following countdown queue entry. This conflict delays the execution of the instruction in the countdown queue by one cycle. Consequently, the instruction will continue to make repeated requests for switchback until either i) it finds an available path, or ii) it reaches the end of the countdown queue at which point it is guaranteed entry into the tail of the main queue. The topology of the Cyclone scheduler queues is such that every instruction will have at least one empty queue entry to move into for the following cycle. In addition, instructions are also guaranteed to eventually reach execution (by traversing the entire length of the queues). As a result, forward scheduler progress is always maintained and no global synchronization is required, which keeps scheduler circuit speeds fast.

Each Cyclone scheduler row is associated with a specific group of functional units. As shown in Figure 25a, instructions switchback to the row in the main queue corresponding to the row they are occupying in the countdown queue. This policy ensures that instructions will find the appropriate functional unit when they reach the execute stage. We found in our circuit analyses that the local nature of switchback control provided significant headroom in circuit performance. As a result, for instructions with multiple functional units available to execute their operations, we permit them to switchback to one of two fixed rows in the main queue (with identical functional units), based on availability of queue space. This policy helps to reduce switchback conflicts and leads to more efficient schedules.

4.2.1.6 Replay and Speculation Recovery

The Cyclone scheduler incorporates a unified mechanism to implement selective instruction replay and mispeculation recovery. Immediately proceeding execution, instructions probe a table of physical register ready bits to check the availability of operands. This check is necessary because of the speculative nature of Cyclone scheduling. Any incorrect schedules, due to factors such as latency misprediction, incorrect memory dependence prediction, or switchback conflicts, may alter the schedule, resulting in instructions possibly entering execution before their operands are available. If the ready bits indicate an instruction's operands are ready, its destination register ready bit is set valid and the instruction continues into execution. In the event an instruction cannot complete execution in its predicted execution latency (for example, if a load instruction misses in the cache when a hit was predicted), the instruction sets its destination register ready bit to invalid. Later instructions that access this invalid register will replay and indicate that their result is unavailable, forcing a cascaded dependence-based instruction replay. All instructions that replay enter the countdown queue with a new predicted latency.

Branch and load/store dependence mispeculations are implemented using a similar mechanism. When an instruction behind a mispeculated instruction is squashed, it is marked as such, and the instruction will continue until it reaches execution, at which point the scheduler will drop the instruction. We avoid flushing the scheduler queue as instructions before mispredicted branches may still be in flight, and we also avoid checkpointing Cyclone queue valid bits as this would make access ports into the scheduler queue necessary.

Mispeculated instructions are identified using *speculation masks*, similar to those in the R10000 [87]. The speculation mask of each instruction is included with the instruction in the Cyclone scheduler queues. When instructions reach the end of the main queue, they probe the speculation state. If the table indicates that the instruction has been squashed, it is dropped. For all experiments, we use a five bit speculation mask. This size mask permits at most 32 speculative paths within the window at once. Instructions requiring additional masks would likely be very speculative and have a low probability of retiring.

When a mispeculation occurs, the instruction pre-scheduler timing table must be updated to reflect that instructions squashed no longer are forwarding values to instructions being decoded. We leverage two observations to simplify recovery of the pre-scheduler timing table. First, their is no strict requirement that the instruction pre-scheduler timing table be correct. The primary motivation for updating the timing table is that it improves the schedule accuracy for instructions after mispredicted branches. Second, we observe that instructions following a mispeculation nearly always arrive at execute with their operands ready, because in long pipelines it will take many cycles for newly fetched instructions to reach execution. During this delay, most instructions in flight will have completed. Simulation of the highly speculative benchmark GCC confirmed this claim, as it revealed that 99.98% of all first accessed logical operands were ready following a mispredicted branch. As such, we can accurately approximate the new pre-scheduler timing table by simply resetting all delay times to zero whenever a mispeculation occurs.

A similar technique is used to recover from load mispeculations. Loads are assigned a speculation mask at decode. In the event a later store arrives after a speculative load completes (revealing an incorrect store-forward), the load's speculation mask and those that

follow it are marked invalid and instructions following the load in the Cyclone queue are squashed when they reach execute. Like the register pre-scheduler timing table, all entries in the store-set timing table are set to zero after a mispeculation.

4.2.2 Methodology

4.2.2.1 Architecture Simulation

We simulated both the broadcast-based scheduler and the Cyclone scheduler on two different pipeline configurations. First, we simulated a machine with a width of 4 throughout the pipeline, from fetch to commit. Second, we simulated a machine with a width of 8 throughout the pipeline. In addition, we simulated the Cyclone scheduler with an issue width of 8, but leaving the fetch and commit rates at 4 instructions per cycle. The number of functional units was kept constant across all configurations. The processor had 5 integer units, 2 of which were capable of multiplication/division, and 3 FP units, 2 of which were capable of multiplication/division/square root, and 4 memory ports. FU latencies varied depending on the operation, but all FUs, with the exception of the divide units, were fully pipelined allowing a new instruction to initiate execution each cycle.

The Cyclone simulator models the architecture discussed in section 2. The Cyclone main and countdown queues were fixed at length of 8 which gave the Cyclone an overall loop length of 16. The underlying ROB was varied between 64 and 256 entries. Load/store dependencies are checked within a 32 entry load/store queue.

Our baseline broadcast-based configuration models the current generation out-of-order processor microarchitecture, with parameters outlined in Chapter 3. We modeled

machines with instruction window sizes ranging from 16 to 128 and ROB sizes ranging from 64 to 256 instructions. All configurations had a 32 entry load/store queue.

In order to reduce the effects of unknown stores and harness dependence information between stores and loads, a store-set predictor [14] is included in both our baseline and our Cyclone configurations. The store-set predictor has 128-entries and is 4-way set-associative. The store-set predictor will find links between loads and sourcing stores allowing loads to speculatively execute before unresolved stores ahead in the load-store buffer.

4.2.2.2 Circuit Timing Methodology

To get a full understanding of the consequences of our design decisions, the circuit characteristics of the different scheduler structures must be examined. The circuit delays for the Cyclone scheduler were calculated using the SPICE design flow we discussed in Chapter 3. The circuit delays for CAM-based scheduler windows used in this study were derived from the models used to evaluate the tag elimination mechanism.

4.2.2.3 Area Estimate Methodology

To examine the area footprint of different scheduling designs, register bit equivalent (RBE) areas were computed, as we describe in Chapter 3.

4.2.3 Impact on IPC

The IPC extracted by the Cyclone scheduler is consistently below those produced by the traditional broadcast-based scheduler. The most substantial cause (direct or indirect) of IPC loss is switchback conflicts in the Cyclone. If an instruction wishes to cross from the replay queue to the main queue, it cannot do so if another instruction is currently occupying that slot. This will cause the instruction to cross at a different point which will delay its

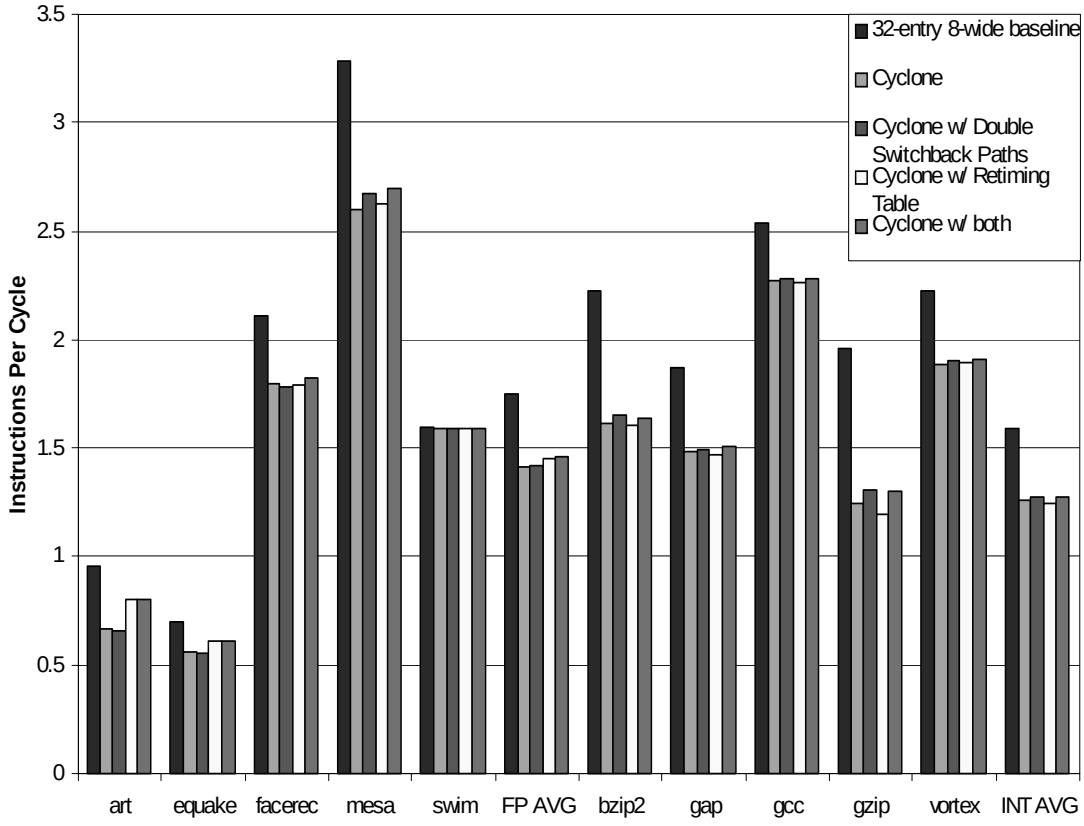


Figure 26. IPC effects of Cyclone optimizations. All configurations have an issue width of 8.

arrival at the execute stage by at least one cycle. This may not significantly impact the execution of the instruction in question, but it can disrupt the execution of the descendents of that instruction. When parent instructions fail to execute as scheduled, children that are scheduled for that parent's completion time may arrive early which will force the child instruction as well as all of that child's descendents to replay. Not only are these instructions now forced to replay, but they are also now consuming slots at the tail of the replay queue into which the decode-stage is trying to insert new instructions.

To mitigate these effects, we tried several different approaches. First, because the Cyclone structure scales well with issue width, we made the queues wider in hopes of

reducing the number of conflicts. In this case, there were still cases where conflicts would occur due to there being a valid entry in the only row that an instruction was allowed to jump to. Because we determined that the switchback logic was far off the critical path, we gave each entry of the replay queue switchback paths to two different row entries in the main queue.

Also, in an effort to improve the accuracy of the Cyclone schedules, we experimented with a Cyclone design which includes variable length dynamic replay. This design adds a retiming table to the register read stage of the pipeline. As instructions leave the main Cyclone queue, they probe the ready bits of their operands. If any operands are unavailable, the retiming table will indicate the number of cycles until the operand is available for use. The instruction then takes the maximum delay of all of its unavailable operands, adds its latency to the result, and then stores this value into the retiming table at the index of its destination register. Finally, the instruction enters the replay queue with a latency equal to the maximum of its unavailable operands. In the baseline system, all replays are signaled with a single ready bit, and the new latency is set to one cycle, meaning that replaying instructions immediately attempt to cross back over to the main queue.

The effects of all these optimizations are shown in Figure 26. Using the double switchback logic, the average IPC was only slightly (~1%) higher, with some benchmarks, like *gzip* seeing as much as 5% improvement. Also, simulation shows that the finer-grained variable-length replay support provided little extra scheduler throughput on most benchmarks. However, *art* and *equake* were 20% and 10% higher, respectively. Given the added area necessary to implement the multi-ported retiming table (approximately 100608 RBEs

Table 3. Critical path latencies calculated with SPICE for different scheduler configurations

Config	Timing (ps)	Config	Timing (ps)
Cyclone	193		
16-entry/4-wide	284	16-entry/8-wide	345
32-entry/4-wide	349	32-entry/8-wide	448
64-entry/4-wide	466	64-entry/8-wide	671
128-entry/4-wide	775	128-entry/8-wide	1243

Table 4. Component breakdown for scheduler and register file area. Areas are in Register Bit Equivalent (RBE)

Config	Scheduler	Reg File	Total
Cyclone 8-wide	16682.4	338504	355186.4
CAM 8-wide 64-entry	143527.7	338504	482031.7
Matrix 8-wide 64-entry	58510.1	338504	397014.1

for an 8-wide configuration), this small additional precision is not likely to be worth the complexity cost for a real design.

4.2.4 Circuit Speed

The SPICE circuit timing results are shown in Table 1. The critical path for the Cyclone runs through the prescheduling logic. It consists of two 8-bit MAX operations, and one 8-bit addition. The switchback logic was also simulated, and it was determined to be well off the critical path. This was because all communication was only to each entry's neighbors and there was only a small amount of logic. The Cyclone also benefits from not needing any selection logic, since it issues one instruction each cycle from every row.

4.2.5 Complexity Tradeoff

There are many different factors to take into account when evaluating scheduler designs. The primary goal is high instruction throughput. This can be accomplished either through high IPC or through low-complexity logic which can be run at a higher clock

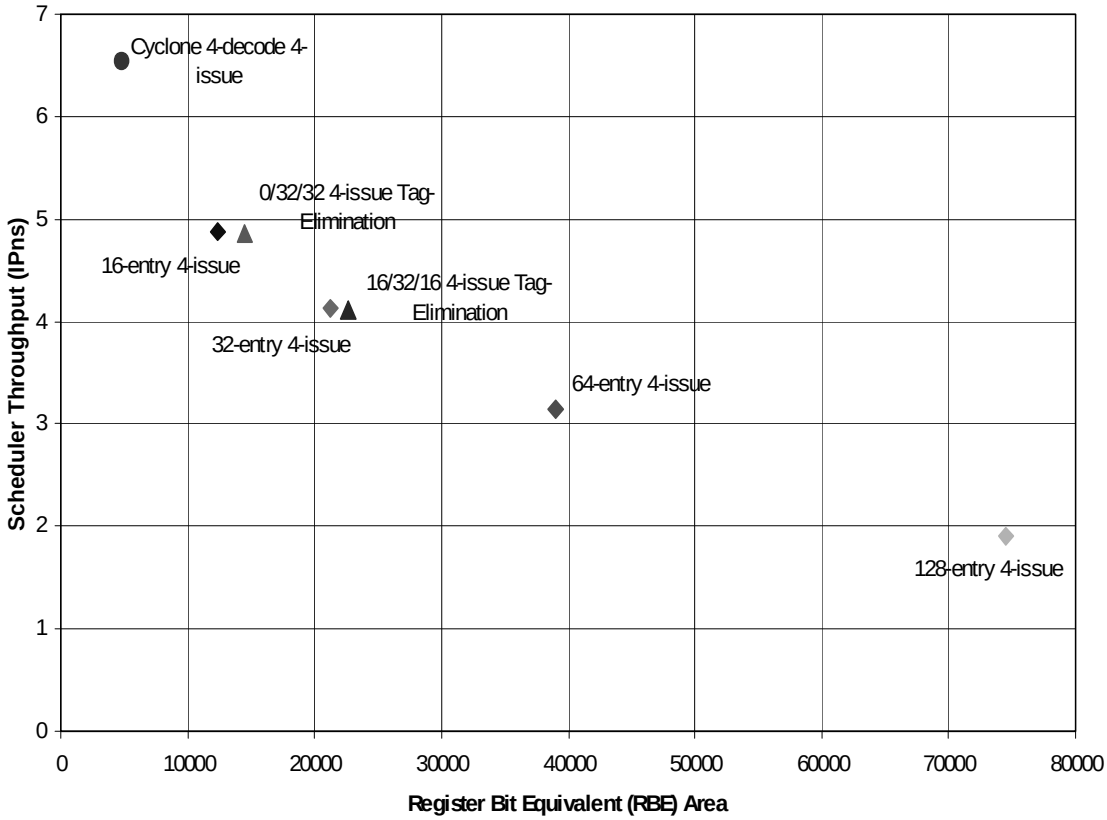


Figure 27. Performance and area for the 4-wide scheduler design space. The most optimal designs are those above (higher performance) and to the left (lower area use) of other designs.

speed. A good scheduler design must also take into account its total power consumption. This factor is tightly related to both the chip area of the design of its circuit complexity.

The Cyclone scheduler takes advantage of this relation by providing a design that is much smaller and less complex than a broadcast-based window. First, because all signals in the structures are local, the throughput is increased due to much faster logic speeds, at the expense of decreased IPC. Also, the Cyclone scheduler structure has a much smaller chip footprint than a broadcast-based scheduler. For example, an 8-decode, 8-issue

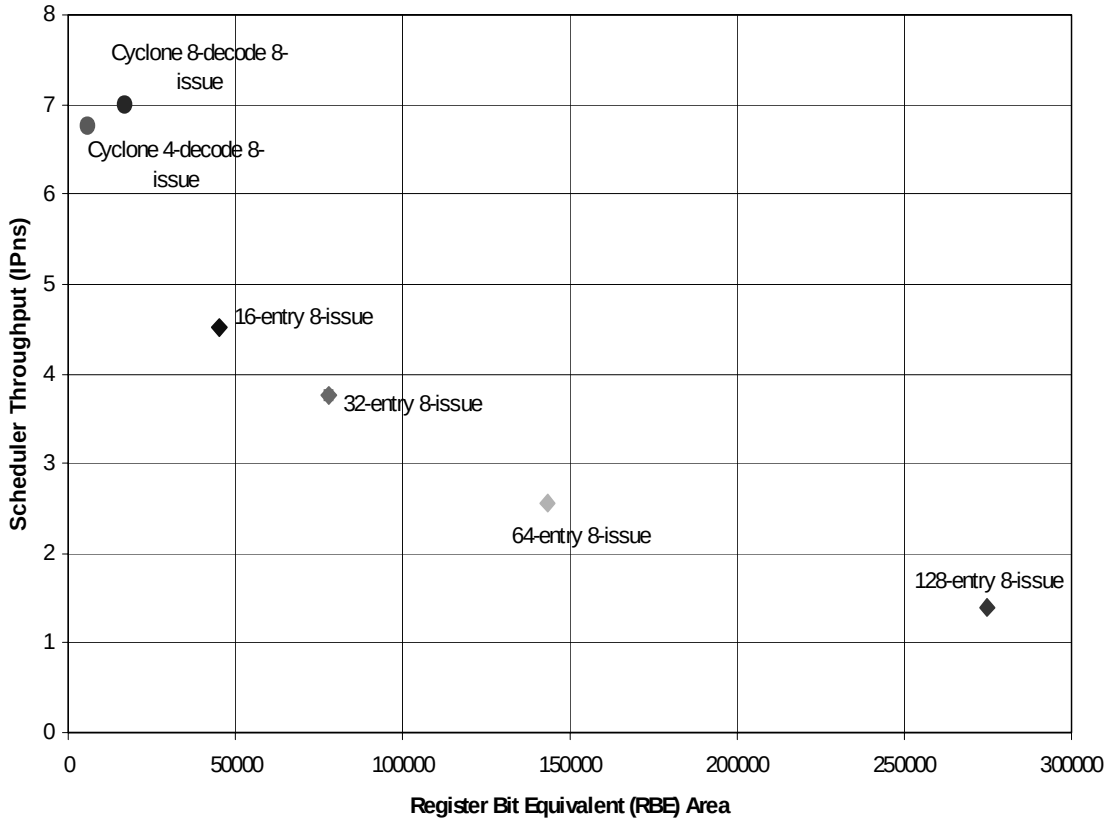


Figure 28. Performance and area for the 8-wide scheduler design space. The most optimal designs are those above (higher performance) and to the left (lower area use) of other designs.

Cyclone queue takes up approximately 12% of the area of a 64-instruction 8-issue CAM scheduler, and approximately 28% of the area of a similarly sized matrix scheduler.

Both of these advantages are seen in Figure 27, Figure 28, and Figure 29, which analyze the tradeoffs between throughput and area. Throughput is presented in single-stage instructions per nanosecond (IPns). We examined both broadcast-based, tag elimination, and Cyclone-style designs, varying instruction window size (for broadcast designs) and issue width. Figure 27 compares 4-wide execute configurations, while Figure 28 examines 8-wide configurations. Because the size of the register file changes dramatically with the

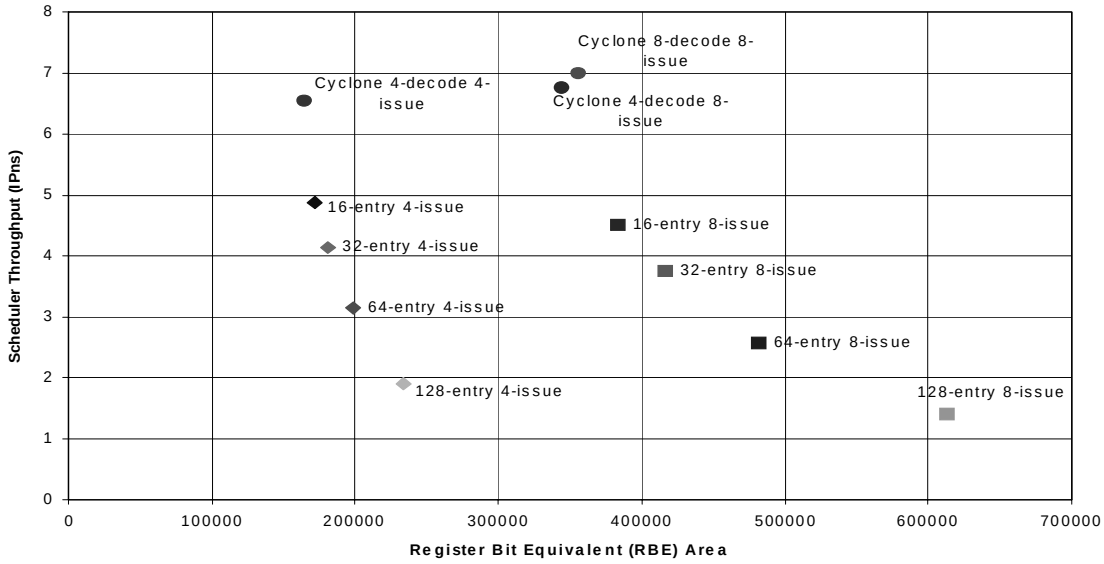


Figure 29. Performance and area overview. Designs shown are for issue widths of 4 and 8 and include register file area. The most optimal designs are those above (higher performance) and to the left (lower area use) of other designs.

number of ports, its total size is a large factor when making decisions on scheduler design. Figure 29 presents the design space for both widths, with the register file area included. The breakdown of the scheduler and register file area is given in Table 2. Overall, the Cyclone scheduler's simplicity and lack of global control allows for fast clock rates and high throughput, but in far less area than same-width conventional designs.

4.3 Summary

Traditionally, there have been two primary instruction scheduling mechanisms, both of which have their respective problems. Compile-time scheduling suffers from less than ideal schedules due to lack of information about runtime events. Dynamic schedulers, while able to generate higher quality schedules, are typically implemented with complex broadcast circuitry that can become a bottleneck in the instruction pipeline. We have introduced the Cyclone scheduler, which draws techniques from both these approaches to

achieve schedules that rival that of other dynamic schedulers without the need for complex wakeup and selection logic. The Cyclone scheduler relies on a simple one-pass scheduling algorithm to predict the time when instructions should execute. Once decided, this schedule is implemented with a timed queue structure that additionally supports efficient selective replay in the event of an incorrect schedule. Even though the Cyclone scheduler design possesses no global communication to slow clock speeds, it rivals the instruction throughput of similarly wide broadcast-based dynamic schedulers, taking up a much smaller chip area.

CHAPTER 5

VIRTUAL GLOBAL CONTROL

As was stated in Chapter 1, it is our goal to find and evaluate complexity-effective alternatives to the global control mechanisms which have high communication complexity. In Chapter 4, we addressed dynamic scheduling, a global control signal with a highly-capacitive load. In this chapter, we find methods to mitigate long-wire problems, specifically those that arise from certain global control signals.

5.1 Sending Control Signals Over Multiple Cycles

As with other trade-off optimizations that cross the circuit/architecture boundary, removing control signals from the critical path involves a comparison process using information from both fields of design. From the architectural side, we need to know the performance impact (on instruction throughput per cycle) of making a control signal take multiple cycles. In the circuit realm, we need to know the savings or loss in both clock period and power usage. In the following sub-sections, we present a process that can be used to make decisions about pipelining chip control signals.

As part of this process, we present two methods for efficiently allowing control signals to use multiple cycles for transmission. The first method, micro rollback, removes the

necessity for inter-stage stall signals to arrive in a single cycle to preserve stall semantics. The second method is a distributed technique based on CounterFlow [71] pipelining, which processes a flush event over multiple cycles using only nearest neighbor communication.

5.1.1 Identifying Problematic Signals

Before a designer attempts to craft solutions, they must first know where these problem exists in the chip. Given a processor in the later stages of physical design, the slowest paths on a chip can be estimated using CAD tools. This evaluation would likely come fairly late in the logic design cycle, as it would give more accurate timing results and the data can be mined from standard timing evaluation runs. The most important data to be collected includes the delay of a set of the longest paths, and the architectural significance of the signals to which those delays correspond.

The delay information can be used in gauging the overall effectiveness of applying an optimization to remove the circuit from the critical path. For example, if designers optimize the longest path in a design, only to have the next-longest path be only infinitesimally shorter, the speedup benefits of the optimization are not likely to be worth the effort.

The information on the signal's logical significance is needed to determine what effect it might have on other related data and control signals, including back-up overheads and architectural slowdown. It is also important in determining the signal's frequency of use.

5.1.2 Methods for Virtualizing Signals

Some of the longest wires in processors occur in *recovery* logic, which is responsible for correcting the repercussions of mispeculations throughout the depth of the pipeline.

Flush logic removes or invalidates instructions from the pipeline that the processor speculated on incorrectly. The most obvious example of flush logic is the pipeline flush that occurs when a branch instruction is mispredicted. In this case, all instructions that follow the branch must be invalidated, and fetching resumed at the correct location. *Stall* logic causes instructions in the pipeline to stop their movement to the next stage so that some dependency further ahead in the pipeline can take extra time to resolve. A simple example of this is the "Load-Use" stall from the standard DLX pipeline [27].

The high complexity of recovery logic comes from the need to tell many stages of the pipeline that some event has occurred. Long wires are needed to distribute this information throughout the chip. Further, the loading at the end of these wires is not small, due to the need to send the signal to several different locations. These two factors combine to make the broadcast of recovery signals very slow.

5.1.2.1 Multicycle Stall Logic

Figure 30 illustrates the simple single-cycle approach to pipeline stalling based on global clock gating. In the event that any stage detects a reason to stall, preceding stages of the pipeline are stalled by gating the next global clock edge. After the stalling instruction resolves, the signal is released, and the clock resumes to the affected stages.

Due to the long wires and large number of recipients involved, it may not be possible to implement global clock gating without significantly impacting processor cycle time. The stall signal must propagate from the error producing stage to all other stages within only a small amount of time. In the Razor design discussed later, this problem is exacerbated by the small amount of time, less than half of a cycle, that the error signal has to make this transition.

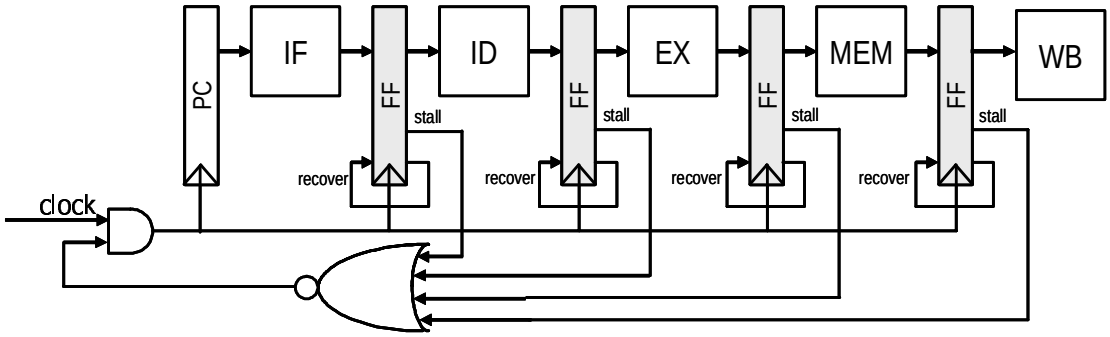


Figure 30. Pipeline stalling using global clock gating

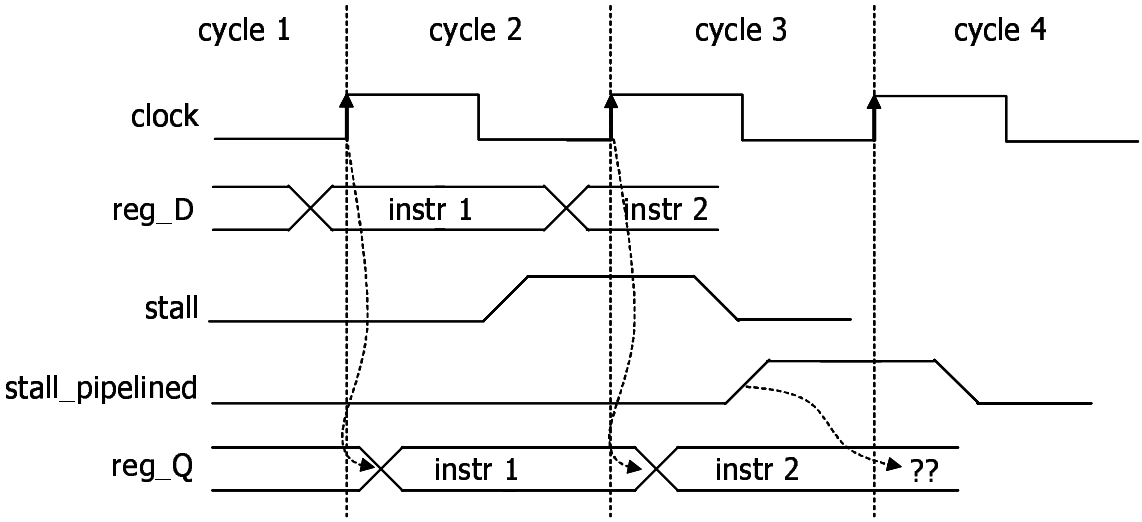


Figure 31. Timing diagram for pipelined stalls

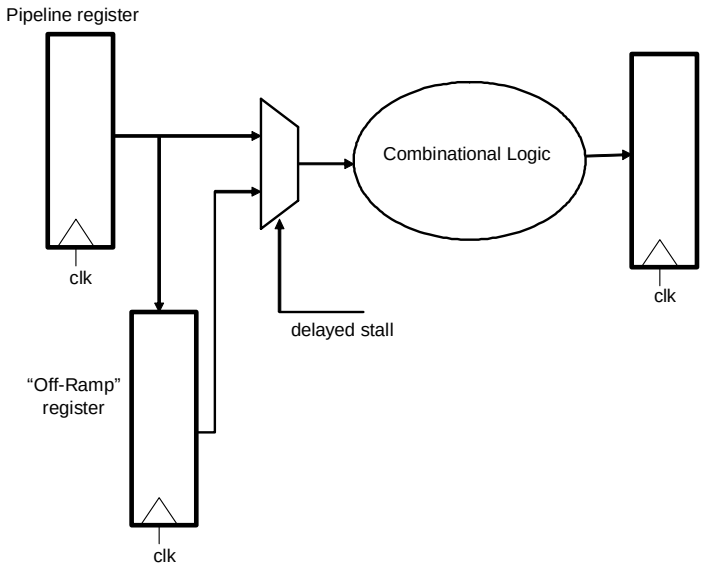


Figure 32. "Off-Ramps" for delayed stall signals

The most obvious answer to the problems with this signal, from a circuit designer's standpoint, is to implement it in multiple cycles. While simply pipelining a stall signal seems a reasonable enough idea, the extra delay would cause disasterous results in the architecture. The problem is illustrated in Figure 31. During cycle 2, the bits for instruction 1 are being held in the flip-flop while it is being processed by the next logic stage. Normally, if a stall is signaled from a later stage, the bits for instruction 1 would be held in the register until the stall was finished. If the stall signal is pipelined, however, its arrival during cycle 3 is too late to stop the bits for instruction 1, which have already been replaced by instruction 2. While instruction 1 may have moved on correctly, it is highly likely that it, or one of the other instructions in the stall shadow, received incorrect data or was not latched in a succeeding stage, resulting in the complete loss of an instruction.

One partial solution to the difficulty of global stalling is the use of "off ramps"[1] to save previous cycle instruction bits and allow stall signals more time to propagate across the chip. As shown in Figure 32, the ramp consists of an extra register backing the pipeline register. Each cycle, the ramp register saves the value that was in the pipeline register the cycle before. By keeping this value around an extra cycle, any stall signals needing to signal this stage now have an extra cycle to reach it. When the signal is received, the values going into the next stage will be taken from the ramp register instead of the normal pipeline register.

This technique preserves the same instruction timings while giving the stall signals more time to propagate. However, this scheme is only applicable when one extra cycle is

1. The "off ramps" described have a similar purpose to the emergency ramps seen on mountainous highways, which are intended for large trucks that are not able to slow themselves quickly enough to avoid burning out their brakes.

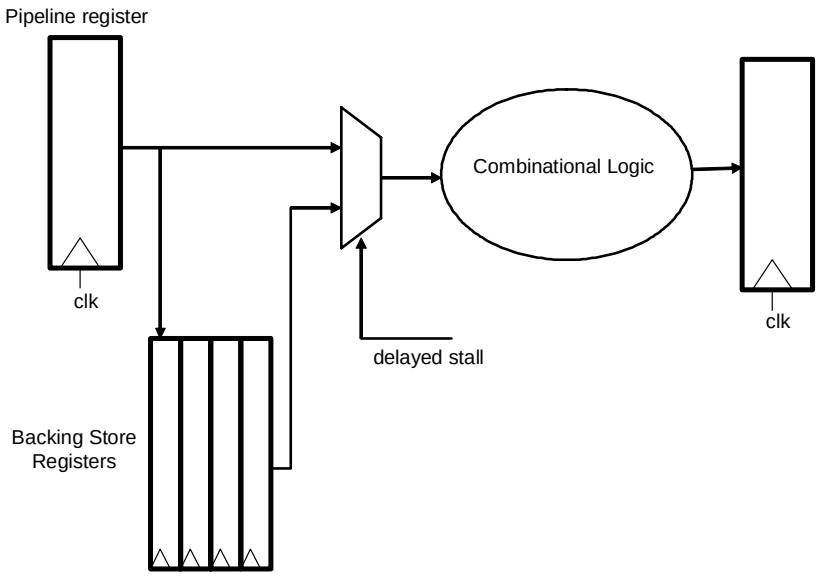


Figure 33. Mechanism for *VGC* micro-rollback

sufficient time for the stall signals to propagate to all stages. While this may work for smaller designs, many studies [2][31][78] have shown that, in future processes, it may take several cycles for signals to propagate across chips, even given low loads and optimal repeaters.

An extension of this technique, *micro rollback*, was proposed and discussed at length by Tamir, Trembley, and Rennels [80][82][83] as a method to quickly recover from transient faults that occurred during execution. After examining this design, it is clear that its ideas would apply easily to stall logic as well.

As is shown in Figure 33, micro rollback keeps a FIFO queue of length *N* backing each register. Similar to the "off ramp" technique, the value in the pipeline register is saved each cycle to the backing storage. In the case of micro rollback, it is pushed into the queue. Therefore, the queue will hold the pipeline register values from the last *N* cycles, allowing a rollback of more than just one cycle. In essence, the processor is making a dis-

tributed checkpoint of state each cycle. On a stall event, the stall signal can be given up to *N* cycles to propagate to all the necessary stages, or the stall signal could reach some stages in different cycles, giving a different rollback latency for each.

The drawback of this technique is the overhead associated with the extra backing storage elements for all registers that need to stall for a late signal. Adding more registers could significantly increase area and power overheads, especially in deeply pipelined designs. It is therefore obviously important to design the backing storage circuitry carefully, to minimize power, both dynamic and static, without affecting the timing or drive of the baseline register. This can be accomplished through careful sizing and process techniques such as using high V_t transistors.

In an effort to analyze the ballpark of these overheads, we modeled a flip-flop with two different 1-deep backing store designs using SPICE with parameters from an IBM 0.13µ process.

An initial design for this structure is shown in Figure 34. An extra flip-flop (A) is attached to the second set of cross-coupled inverters of the baseline flip-flop (B). A 2-input pass-through multiplexer (C), controlled by the stall signal, is then placed between the storage elements and the output drivers. To protect against hold time violations, an extra buffer is inserted between the main flip-flop and the backing flip-flop. Under normal circumstances, the multiplexer passes the output value from the base flip-flop, preserving the same operation as in a default design. However, if the stall signal is asserted, the multiplexer selects the data from the extra flip-flop, passing out the values that the baseline flip-flop had held the cycle before. The resulting effect is that of a 2-cycle stall.

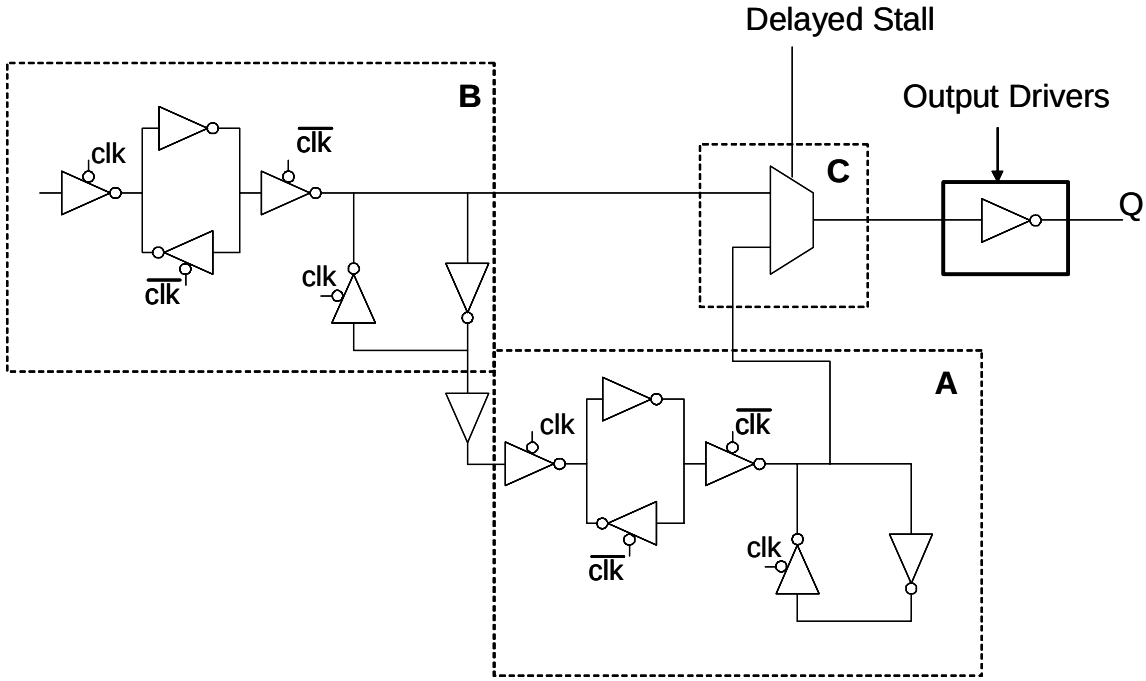


Figure 34. Rollback register design 1

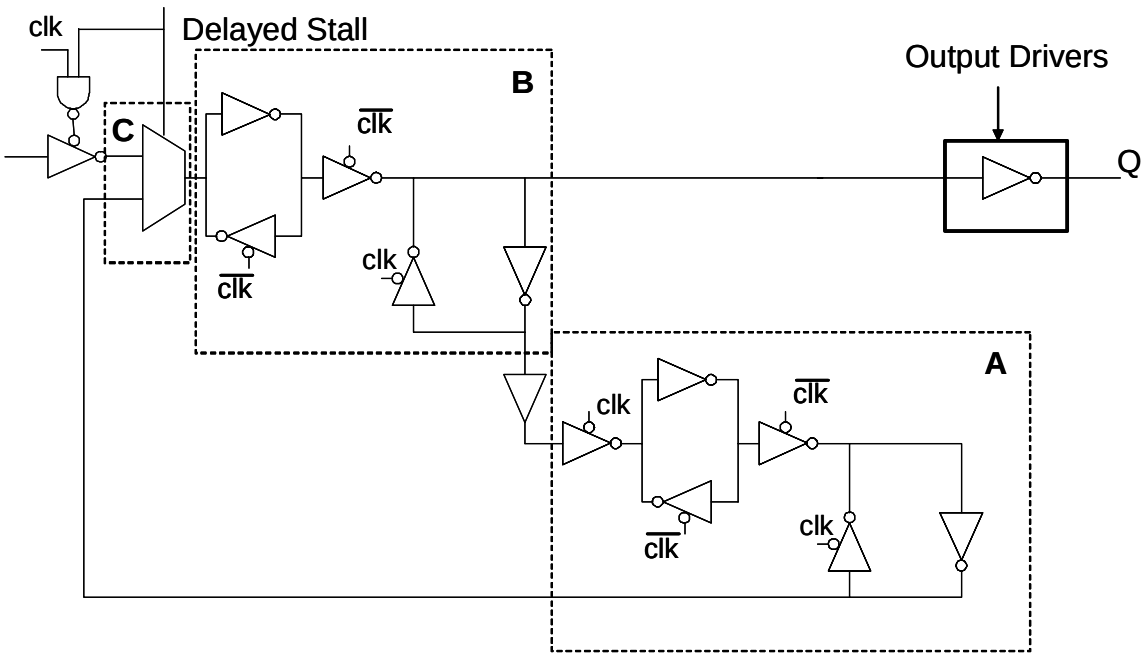


Figure 35. Rollback register design 2

To keep the overheads of this design low, the transistor sizings of the backup flip-flop should be reduced as much as possible. However, the direct exposure of the extra storage to the output drivers of the circuit curtails our ability to greatly modify the gates, due to the need for this extra element to provide a similar drive strength as the original flip-flop. This also creates a separate balancing problem, as the additional loading on the output of the main slave latch will reduce its ability to drive the output as well. Further, because the multiplexed output must be driven out for most of a cycle, the delayed stall signal controlling the multiplexer element must be held active for a long period of time, making it extremely difficult for that signal to meet acceptable timing constraints. Finally, in addition to controlling the multiplexer, the delayed stall signal needs to gate off the backup flip-flop clock, allowing the data to stay in place while it is driven out.

A second organization, shown in Figure 35 (with the same labelings as Figure 34), was designed to remove the external exposure of the backup storage and resolve the need for a long hold time. The multiplexer is moved to route into the feed-forward path of the main storage and chooses between the standard D input value and the value stored in the backup register. In this setup, the output of the primary flip-flop is always sent to the output drivers. In the case of a stall, the value from the backup storage is routed back into the primary register via the mux during the negative edge. In the next cycle, the main register drives out the saved data, resulting in the effect of a 2-cycle stall.

Table 5. Power Consumption of registers with micro rollback

Flip Flop	Flip-Flop Energy	Relative Energy	Relative Area
Default	24.74 fJ	1.0	1.0
Micro Rollback	28.17 fJ	1.1387	1.210
Rollback w/ high V_t	27.73 fJ	1.1209	1.210[a]

a. A circuit segment with a differet threshold voltage requires laying out an extra well area around the transistors using the higher threshold. Therefore, while using high-Vt transistors does not change the area by our metric, the dual-threshold circuit will likely have a slightly larger chip area.

In contrast to the first design, the backup storage is allowed very generous timing and sizing constraints in this model due to its removal from the drive path. This allows us to reduce the size of these elements as much as possible. Also, to reduce leakage power, it is reasonably easy to use high V_t transistors for the entire backup storage. Besides reducing power overhead, these changes greatly reduce the extra burden on the main register's slave latch, allowing it to stay largely unchanged from a default library design. In this second design, the constraints on the stall signal are also greatly loosened, needing for it to only meet the setup and hold time for the main flip-flop. To enable this setup to support multi-cycle stalls (stalls which last longer than 1 cycle in length), the delayed stall signal would also need to gate off the input to the slave of the backup flip-flop, preventing the one cycle of "garbage" data from corrupting the saved value during the first cycle of stalling.

This final design, with a small inverter at its output, was implemented in an IBM 0.13µ process and analyzed using HSPICE, with energy results shown in Table 5. The measurements reflect both the 1-to-0 and 0-to-1 transitions of the flip-flop over 4 ns cycles, along with the static energy dissipation during non-switching cycles. Given reasonable sizings under the constraints discussed above, the backup register adds approxi-

mately 13.9% overhead in energy. In addition, using high V_t transistors (IBM library's "lppfet" and "lpnfet") for the backup storage further reduces this overhead to approximately 12.1%.

One difficulty in designing a system with micro rollback is determining how many cycles worth of data the backing store must hold for each stage. First, it must be determined what the maximum possible latency is for a stall signal to propagate across the chip. Analysis must also be done to see if there is benefit (or complication) from "staggering" the signal such that some stages receive it earlier than others. It is obvious that such decisions will widely vary between different chip designs. As a case study, we studied an implementation of the micro rollback mechanism to handle the error-signal stalls in the Razor system and present this in Section 5.2.2.

5.1.2.2 Virtual Flush and Recovery Logic

In contrast to stall logic, flush logic does not need to preserve the previous state of the instructions in pipeline stages that it affects. The pipeline recovery mechanism must guarantee, however, that register and memory state is not corrupted with an incorrect or invalid value from a flushed instruction. In this section, we highlight a possible approach to implementing low-complexity pipeline error recovery based on counterflow pipelining.

A flush and recovery system in a processor has several major tasks to accomplish. First, it must invalidate all instructions that are caught under the flush event and prevent them from committing anything to the permanent processor state. Second, after the flush is completed, the system needs to get the processor to restart execution in the correct location. This is usually accomplished by remembering some context of where the processor was in the execution of the program at the time the flushing instruction entered the pipe-

line. In out-of-order processors, there may also be the additional task of not interfering with the execution of instructions that aren't affected by the flushing event. Most commonly, this is addressed by assigning each instruction some kind of related "mask" [87] of bits that identifies how it should react to flush events generated by certain instructions.

The most difficult portion of these tasks is the need to get flush event information to every preceding stage as quickly as possible. This ends up being a major broadcast event and may also additionally include long-distance signal travel across portions of a chip, depending on the size of the flush and the design's floorplan. It may therefore prove useful to explore distributed, multi-cycle approaches to accomplish the flush system's tasks.

Again, the most obvious solution to this complexity problem is to pipeline the signal. Unlike stall signals, flushing does not require the preservation of any of the flushed data, and so does not have the same pipelining difficulties. Also, because flush signals are usually associated with "bad" chip events, such as mispredictions, architects already do their best to minimize the number of times the signal is used. This means that any additional pipelining cycle penalty would also be minimized in this effort.

The more pressing issue is then deciding how these signals should be pipelined. Simply giving the signal multiple cycles to broadcast to all flushing stages may be effective on a small scale, but given the scalability issues of broadcast signals discussed in Chapter 2, it may be more advantageous (and interesting) to explore ways to perform flushing while only using local (point-to-point) communication.

The use of decentralized (local-only) communication is the central idea of the counterdataflow architecture [72][71][49]. In the counterflow pipeline, instructions and data flow

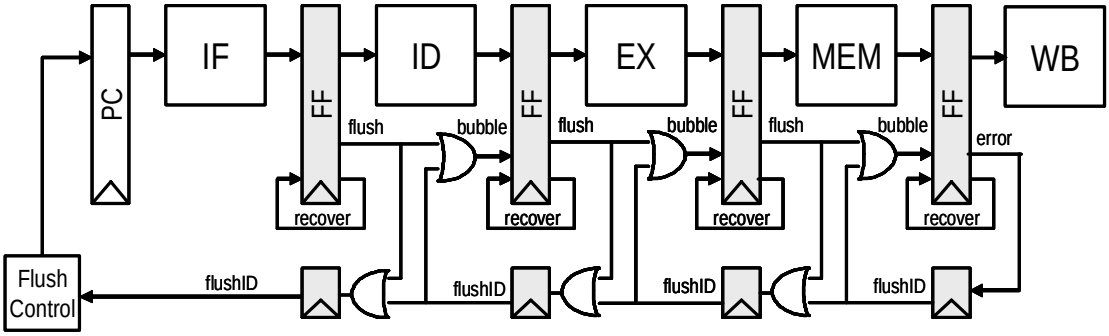


Figure 36. Pipeline recovery using counterflow pipelining

in opposite directions. When instructions that are waiting to execute pass input operands, the data needed to execute is captured by the dependent instruction.

We have designed a fully pipelined flush and recovery mechanism based on these counterflow pipelining techniques. The approach, illustrated in Figure 36, places negligible cycle timing constraints on the baseline pipeline design at the expense of extending pipeline recovery over a few cycles. Instead of data flowing in the opposite direction of instructions, our scheme sends flush information “upstream” in the pipeline, invalidating instructions that are met or passed on the way. When the signal reaches the first stage of the pipeline, the signal stops and initiates the re-fill of the processor.

When a flush event is detected, two specific actions must be taken by the originating stage. First, if the instruction in the generating stage is part of the flush, it must be nullified. This action is accomplished using the *bubble* signal, which indicates to the next and subsequent stages that the pipeline slot is empty. Second, the *flush train* sends the flush-triggering instruction’s mask or stage number back to the previous stage, and it begins propagating this mask in the opposite direction of instructions.

At each stage visited by the active flush train, the corresponding pipeline stage and the one immediately preceding are replaced with a bubble if their instructions are a part of the flush event. (Two stages must be nullified to account for the twice relative speed of the flush signal versus the main pipeline) When the flush signal reaches the start of the flushed area of the pipeline, the flush control logic restarts the pipeline at the next correct instruction, based on information sent in the mask or stage number. In the event that multiple stages experience flush events in the same cycle, all will initiate recovery but only the flush that is the deepest into the pipeline will be completed. Earlier recoveries will be flushed by later ones.

In smaller designs, where this stage-by-stage approach could be considered “overkill” for the amount of complexity of the signal, compromise designs may be more suitable. It can be easily seen that the relative speed of the flush train could be doubled or tripled, taking fewer cycles to traverse the pipeline, but also requiring it to communicate with double or triple the number of instructions per cycle. The further the design moves in this direction, however, the closer it comes to the original broadcast model.

Unlike micro rollback, using a counterflow flush mechanism requires very little area and power overhead. Only a small number of registers, used to split up the propagation of the flush signal, are added. It is likely that the reduced cycle-time requirements and smaller loads in each stage will allow a designer to more than make up for the overhead in power savings on the signal wires themselves.

5.1.3 Quantifying Gains and Losses

Once we have identified problematic signals and chosen one of these methods as a possible solution, an analysis must be done to quantify the impact of the changes on sev-

eral aspects of the design. Architecturally, it must be determined what the performance impact is of adding extra wait cycles under certain circumstances. As mentioned previously, it is important to quantify the cycle time gained by applying these changes. Finally, the area and power overheads of the optimization need to be examined (for the extra micro rollback registers, for example).

5.1.3.1 Impact on Architectural Throughput

When it is determined what role the critical circuit path has within the architecture of the processor, it becomes possible to simulate the impact of "cutting" that path. The modeling of the optimizations presented in the previous section can be performed in a fairly simple and straightforward manner.

If the counterflow flush approach is used, the performance impact can be modeled most simply by adding *N* cycles to the re-run penalty each time the flush event occurs. *N*, in this context, is the number of cycles the staggered flush takes to get from the flush generating stage to the earliest logical stage that must be flushed. Note that, given certain floorplans, it may happen that the flush signal reaches the earliest logical stage before flushing some of the later stages. In this case, it is still sufficient to begin re-running instructions after only *N* cycles, instead of waiting for all stages to be flushed.

If the micro rollback or off-ramp approach is used, the rollback length *N* needs to be added to the amount of time spent stalling for a given set of stages. Note that the forward movement and subsequent re-loading of pipeline register values does not need to be modeled in detail. For example, if an event would cause a stall in the IF and ID stages of a processor, modeling a 2-cycle micro rollback penalty would simply consist of stalling those stages for 3 cycles instead of 1.

In either case, it is clear that minimizing the performance impact is important in the design. There are two factors which are critical to the overall performance impact, which need to be minimized: the number of extra cycles inserted per event (*N*) and the frequency of the event's occurrence (*k*). The *N* parameter is largely a function of process circuit capabilities and/or advantageous floorplanning. The *k* factor is difficult to alter in most cases, as it is an intrinsic property to the signal which was "chosen" in the first place. If the signal is a highly-used data path (a common path through an adder, for example), it is very unlikely that much benefit could be gained from applying one of these optimizations, because the number of cycle penalties would far outweigh the clock speed benefit.

5.1.3.2 Impact on Circuit Speed

A primary purpose of implementing virtualizing optimizations is to reduce a processor's cycle time, which is accomplished by removing a slow signal from the critical path.

As was discussed earlier in this chapter, when designers identify the slowest signals in a chip, they also should collect information on other long paths in the design. When a signal is singled out for virtualization, it can then be determined what the cycle time margin is between the path and its next-slowest successor. Then, when evaluating the feasibility of optimizing that path, this new cycle time, combined with an adjusted architectural performance measurement, can be weighed against the original performance and cycle time.

Further, this process can be repeated iteratively on each new slowest path, resulting in an overall cycle time gain and an overall architectural performance impact.

5.1.3.3 Impact on Processor Power

A third factor that is affected by virtualization optimizations is processor power consumption. There are several ways in which the amount of energy consumed by a system changes with these structures.

First, when changing the cycle count requirement of signals involving long wire delays, it may greatly reduce the need for strong drivers, large repeaters, and other circuit techniques that would be required to make these signals work under tighter time constraints. These structures generally consume more power and area in order to reduce the time needed to for signals to traverse the distance.

The power consumption of the processor can also be impacted in the other direction. Micro rollback requires the addition of extra storage elements for all pipeline registers in a given stage that need to be backed up. While we showed in a previous section that these extra registers can be made as power-friendly as possible, especially in processes that allow for high-V_t transistor areas, the overhead is still non-trivial.

5.1.4 Goal-Oriented Decision Making

After examining the impact a signal-cutting will have on all of these factors, a designer has the information needed to make a decision on which design point is most desirable. This decision can be made based on the particular processor goal that the designer has in mind, whether it's getting the design under a power budget, providing the highest performance possible, or some middle ground between the two (such as ED^2).

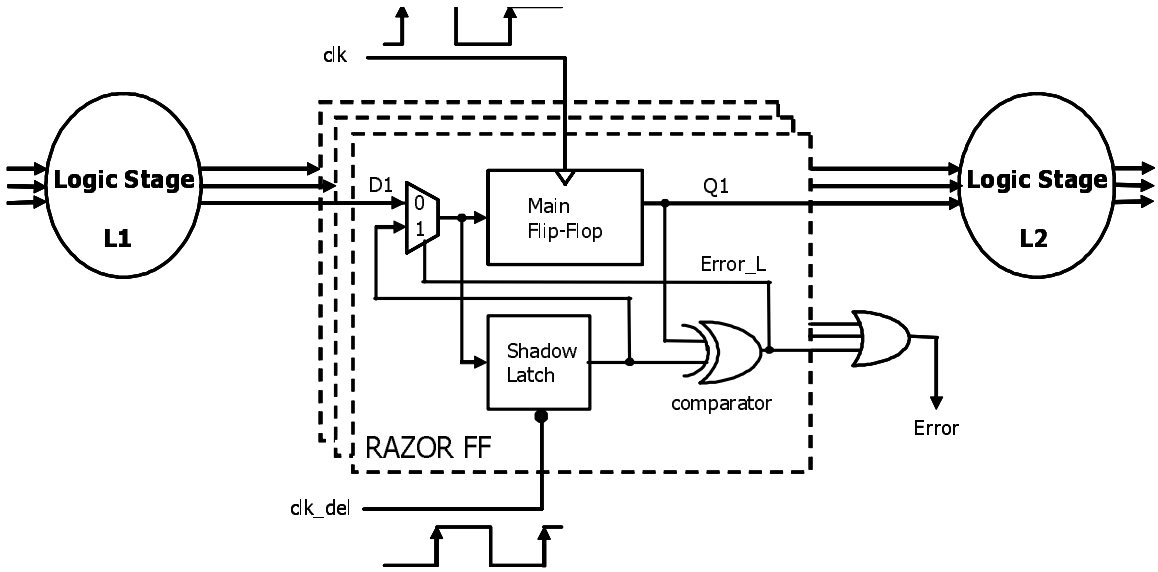


Figure 37. Pipeline augmented with Razor latches and control lines.

5.2 Applying Virtual Global Control to the Razor Prototype Chip

The easiest place experiment with such a paradigm would be using a simple in-order processor, and that will be the focus of this section. As a primary test case, the virtual global control idea will be used to improve the error recovery penalty signal and branch misprediction signal from the chip design discussed in our work on Razor [19]. To find optimizations other than these, examining a real industry design throughout the timing-closure phase could expose other global communication signals that we hadn't considered previously. This sort of analysis, while certainly interesting, is outside both the scope of this work and my personal means.

5.2.1 The Razor System

In a paper in MICRO-36 [19], we proposed a new approach to dynamic voltage scaling (DVS), referred to as *Razor*, which is based on dynamic detection and correction of speed path failures in digital designs. The key idea of Razor is to dynamically detect and correct timing errors in circuits, and then to tune the chip supply voltage by monitoring the

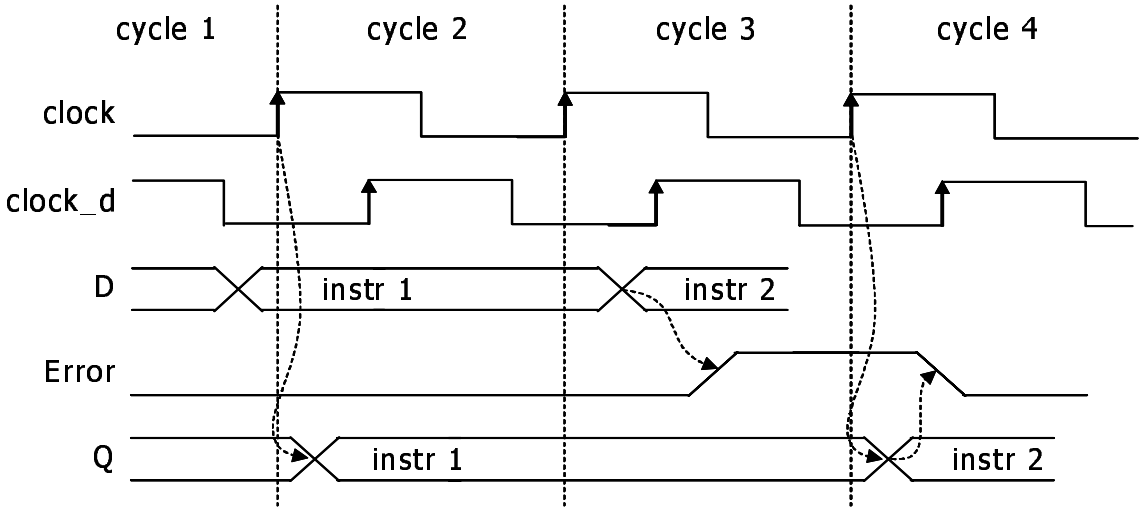


Figure 38. Timing example of Razor operation

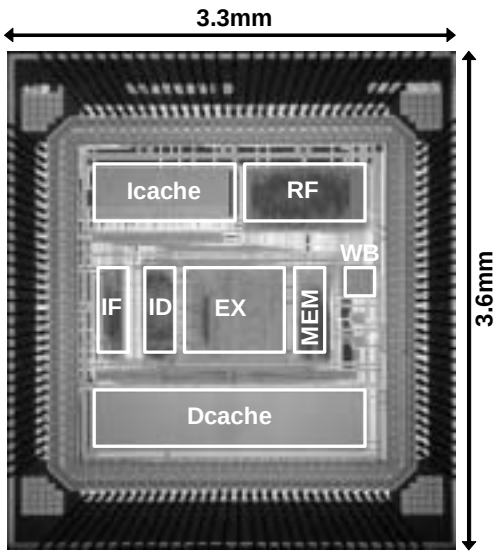
error rate observed during operation. Since this error detection provides *in-situ* monitoring of the actual circuit delay, it accounts for both global and local delay variations and does not suffer from voltage scaling disparities. Because of this, it eliminates the need for voltage margins that are necessary for "always-correct" circuit operation in traditional designs. Conceptually, the key feature of Razor is that operation at sub-critical supply voltages does not constitute a catastrophic failure, but instead represents a trade-off between the power penalty incurred from error correction against additional power savings obtained from operating at a lower supply voltage.

Razor relies on a combination of architectural and circuit level techniques for efficient error detection and correction of delay path failures. The implementation concept of Razor is illustrated in Figure 37 for a pipeline stage. Each flip-flop in the design is augmented with a so-called *shadow latch* which is controlled by a delayed clock. We illustrate the operation of a Razor flip-flop in Figure 38. In clock cycle 1, the combinational logic $L1$ meets the setup time by the rising edge of the clock and both the main flip-flop and the

shadow latch will latch the correct data. In this case, the error signal at the output of the XOR gate remains low and the operation of the pipeline is unaltered.

In cycle 2, we show an example of the operation when the combinational logic exceeds the intended delay due to sub-critical voltage scaling. In this case, the data is not latched by the main flip-flop, but since the shadow-latch operates using a delayed clock, it successfully latches the data some time in cycle 3. To guarantee that the shadow latch will always latch the input data correctly, the allowable operating voltage is constrained at design time such that under worst-case conditions, the logic delay does not exceed the setup time of the shadow latch. By comparing the valid data of the shadow latch with the data in the main flip-flop, an error signal is then generated in cycle 3 and in the subsequent cycle, cycle 4, the valid data in the shadow latch is restored into the main flip-flop and becomes available to the next pipeline stage $L2$. Note that the local error signals *Error_l* are OR'ed together to ensure that the data in all flip-flops is restored even when only one of the Razor flip-flops generates an error.

If an error occurs in pipeline stage $L1$ in a particular clock cycle, the data in $L2$ in the following clock cycle is incorrect and must be flushed from the pipeline. However, since the shadow latch contains the correct output data of pipeline stage $L1$, the instruction does not need to be re-executed through this failing stage. Thus, a key feature of Razor is that if an instruction fails in a particular pipeline stage it is re-executed through the *following* pipeline stage, while incurring a one cycle penalty. The proposed approach therefore *guarantees forward progress* of a failing instruction, which is essential to avoid the perpetual failure of an instruction at a particular stage in the pipeline.



Technology Node	0.18µm
Max. Clock Frequency	140MHz
DVS Supply Voltage Range	1.2-1.8V
Total Number of Transistors	1.58million
Die Size	3.3mm*3.6mm
Measured Chip Power at 1.8V	130mW
Icache Size	8KB
Dcache Size	8KB
Total Number of Flip-Flops	2408
Total Number of Razor Flip-Flops	207
Number of Delay Buffers Added	2498
Error Free Operation (Simulation Results)	
Standard FF Energy (Static/Switching)	49fJ/124fJ
RFF Energy (Static/Switching)	60fJ/185fJ
Total Delay Buffer Power Overhead	3.7mW
% Total Chip Power Overhead	2.9%
Error Correction and Recovery Overhead	
Energy of a RFF per error event	260fJ

Figure 39. Razor prototype chip and vital statistics

In addition to invalidating the data in the following pipeline stage, an error must also stall the preceding pipeline stages while the shadow latch data is restored into the main flip-flop. In the original work, a number of different methods, such as clock gating or flushing the instruction in the preceding stages, were proposed to accomplish this.

The Razor approach also raises a number of circuit related issues. The Razor flip-flop must be constructed such that the power and delay overhead is minimized. Also, the presence of the delayed clock introduces a new short-path constraint in the design. And finally, allowing the setup time of the main flip-flop to be exceeded raises the possibility of metastability. These issues are further discussed in the Razor-specific publications [19][41].

The proposed Razor technique was included as part of a prototype 64-bit Alpha processor design, which was implemented in TSMC's 0.18µ process. This prototype layout was used to obtain a realistic prediction of the power overhead for Razor's *in-situ* error correction and detection. We also studied the error-rate trends for datapath components

using both circuit-level simulation as well as silicon measurements of a full-custom multiplier block. Architectural simulations were then performed to analyze the overall throughput and power characteristics of Razor based DVS for different benchmark test programs. It was demonstrated that, on average, Razor reduced power consumption by more than 40%, compared to traditional design-time DVS and delay-chain based approaches.

The Razor prototype chip is a simple design synthesized from Verilog HDL by the Synopsys design compiler and laid out by Cadence's Silicon Ensemble place and route tool. A layout picture of the original design is shown in Figure 39, along with some properties of the chip.

5.2.2 Updating the Razor System with Micro Rollback

In the original work on Razor [19], we stated that there was an obvious complexity issue with the global control signal which was required to stall the processor when an error was detected. In less than half of a cycle, the error signal would have to propagate through a very high fan-in OR gate and across several stages of the processor, with enough drive to gate off the clock to these stages. To address this complexity issue, we proposed the counterflow flush mechanism which we discussed in section 5.1.2.2.

While that approach accomplishes the goal of removing the critical timing path through this logic, the resulting loss of valid computation that had been correctly done in previous stages is inefficient for both performance and energy consumption. The most natural control option for this situation is not a *flush*, but a *stall*, a realm more suitable for micro rollback.

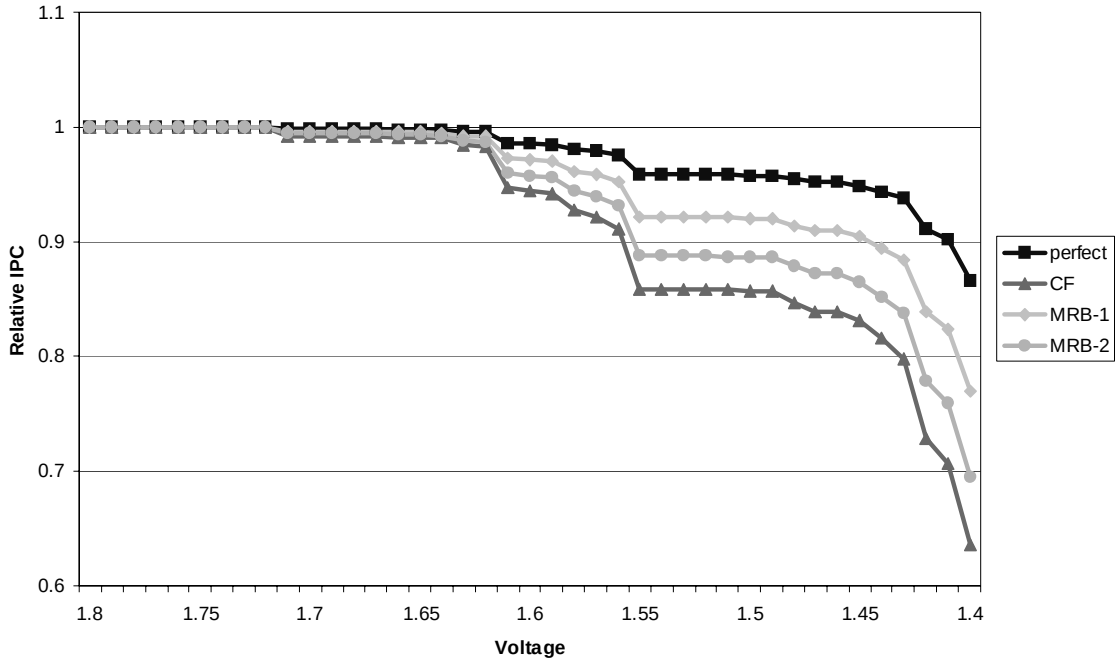


Figure 40. Relative Instructions per Cycle for Various Razor Error Signal Designs

The first impact of switching the recovery mechanism to micro rollback is the change in architectural throughput (IPC). To examine this impact, several designs were simulated: a design that flushes the pipeline on an error, two micro-rollback designs with one and two cycle delays, and a design with a perfect instantaneous clock gated stall signal.

The designs were evaluated using Razor's circuit-aware architectural simulator [41], running a set of SPEC integer benchmarks. The simulator, which closely models the test chip design, uses simulation of verilog models in conjunction with program data to simulate where timing errors occur given certain voltage and frequency information. Details on our simulation infrastructure can be found in Chapter 3.

As would be expected, the performance gap between these techniques changes depending on the error rate of the processor - the rate at which the error signal is asserted. Figure 40 shows the IPC of the 4 design points across a range of run-time voltages. There

is very little difference in performance difference between the designs when the voltage is high and the error rate is low. However, as error rates begin to increase, there is a clear divergence in the different designs' rate of degradation. As would be expected, the techniques with a lower penalty per error signal switch lose performance much more gracefully than the flush design. In the context of a deeper pipeline, this difference would be even more pronounced.

The micro rollback power overhead for the entire processor can be estimated with the equation:

Power Overhead = PO_{FF} x k_{MRB} x P_{FF} ,

where PO_{FF} is the relative power of a flip-flop with micro rollback compared to the baseline, k_{MRB} is the ratio of flip-flops enabled for micro rollback over the total number of flip-flops in the chip, and P_{FF} is the percentage of overall chip power consumed by the flip-flops.

For an implementation of micro rollback on the Razor error signal, all of the pipeline registers in stages previous to the one where the error occurs need to include rollback support. In the case of the prototype design, the latest an error can occur is in the memory stage of the pipeline. Therefore, rollback registers are needed at the end of the fetch, decode, and execute stages, which add up to about 60% of the chip's pipeline registers. Due to the fact that the Razor design doesn't write results to permanent state if an error could possibly occur in front of it, there is no need to add micro rollback to the register file or to memory. (If it had been necessary, both could have been done easily by buffering any writes.)

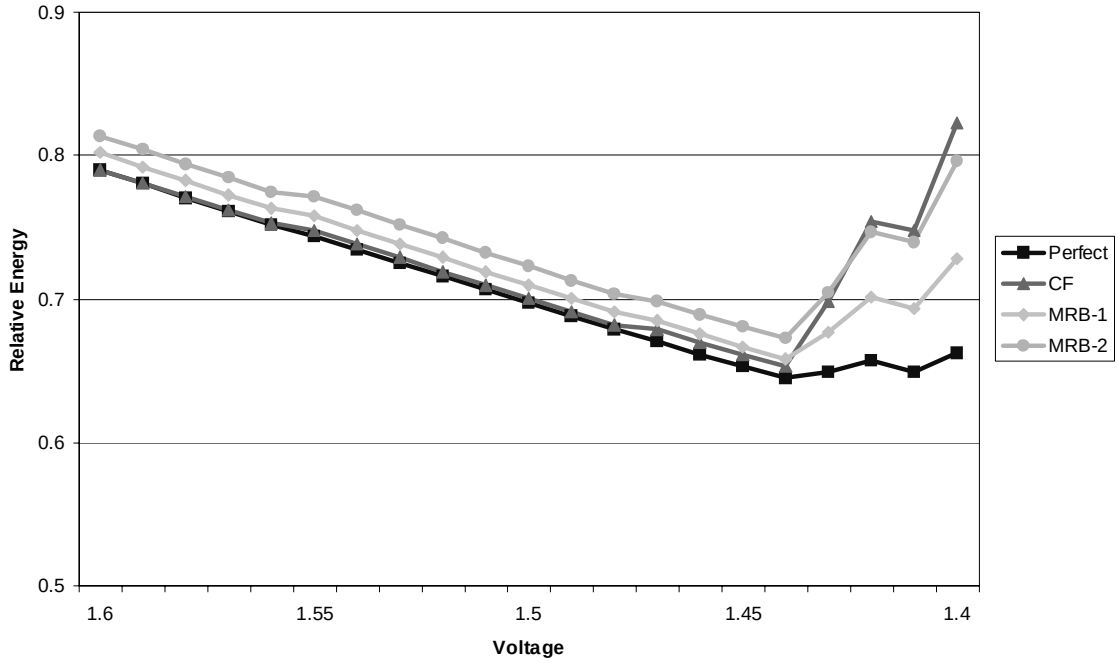


Figure 41. Relative Energy Consumption for Various Razor Error Signal Designs

Using the 12.8% overhead per rollback register from Section 4.1.2.1 and a total flip-flop power share of 20%, the equation above yields a power overhead of 1.5% for a single rollback cycle and 3.0% for a two-cycle rollback.

Figure 41 shows the simulated energy consumption of the processor across a range of voltages. At voltages with error-free operation, their small overhead places the micro rollback designs slightly higher in energy than the baseline or flush designs. However, the higher-penalty techniques quickly pass them by at low voltages.

If the energy curve was a continuous function over voltage, the reduced penalty that micro rollback provides should effectively move the optimal voltage point to a lower voltage. However, because this curve has piecewise-linear factors coming from the discrete number of additional circuit paths failing at each lower voltage step, the large jump in error rate at this point is too much for any recovery mechanism to handle, with even a per-

fect stall losing ground below 1.44 Volts. The 1-cycle micro rollback design does, however, effectively give much more “cushion” beyond the optimal point. This is still beneficial, since simulations of the Razor chip with a control system have shown that the processor will often spend large amounts of time just outside of the optimal zone.

5.2.3 Design Bottlenecks in the Razor Prototype

As a further study into applying our techniques to chip signals, the Razor prototype chip design was examined to find other bottlenecks. In order to find the limitations of the chip, we ran the HDL design aggressively through the synthesis and place/route tool, working towards the fastest clock cycle possible. When this layout was completed, logic and wire timings were given to Synopsys’s PrimeTime tool to compute path timings.

At the end of this analysis, the longest path was found to be the signal triggered when a branch misprediction is detected in the EX stage of the pipeline. This path passed across two stages to the IF stage, where it prompted the processor to change the location of the next instruction fetched from memory, and its estimated timing was 2.96 ns.

The timing differential between this path and the next-longest signal, a datapath through the core of the EX stage, was approximately 0.15 ns. If the branch signal can be removed from the critical path, we could therefore increase the clock frequency by 5.0%.

The CounterFlow style pipeline flush was described in section 4.1.2.2. This system was originally proposed as a basic alternative to the clock-gated stall in the error recovery mechanism of Razor. However, the true purpose of this mechanism is not for stall signals, but to aid in flushing. This design does show promise, therefore, in providing a low-complexity method for pipeline flush situations, such as this one.

By using a mechanism such as counterflow flush, we will suffer penalties in the architectural throughput of the design. As discussed earlier, this decrease in IPC should directly correspond to two factors: the percentage of times the signal is switched during computation, and the number of extra cycles added each time the signal is used.

One of the difficulties in designing a counterflow flush system is determining how many stages to flush in each cycle. Obviously, there is a physical limit to the distance a signal can travel each cycle, but there is a definite instruction throughput advantage in traversing the stages in as few cycles as possible.

Because the Razor test chip has only rudimentary branch prediction (always predict not taken), the frequency of mispredictions is very high - around 52% for the benchmarks ran. Luckily, in most modern systems, a branch misprediction event is not only fairly rare, but chip designers actively attempt to reduce them, since they already carry a stiff penalty. Most commercial processors, even in the embedded realm (such as ARM 10 [3] and 11 [4]), now come with somewhat advanced branch prediction. To contrast, an 8k 2 level predictor is correct for 95% of branches in the same set of benchmarks.

Figure 42 shows the simulated relative IPC for configurations with 1, 2, and 3 cycle flush trains with the default not taken branch prediction of the Razor prototype chip. Using a multicycle counterflow flush, and therefore increasing the penalty to 2 or 3 cycles, decreases the average IPC by 3.3% or 6.4%, respectively.

Figure 43 demonstrates the much smaller IPC penalty the processor suffers with advanced branch prediction. Using a counterflow flush mechanism on this design would only reduce average IPC by 0.2% or 0.5% for a 2 or 3 cycle flush penalty.

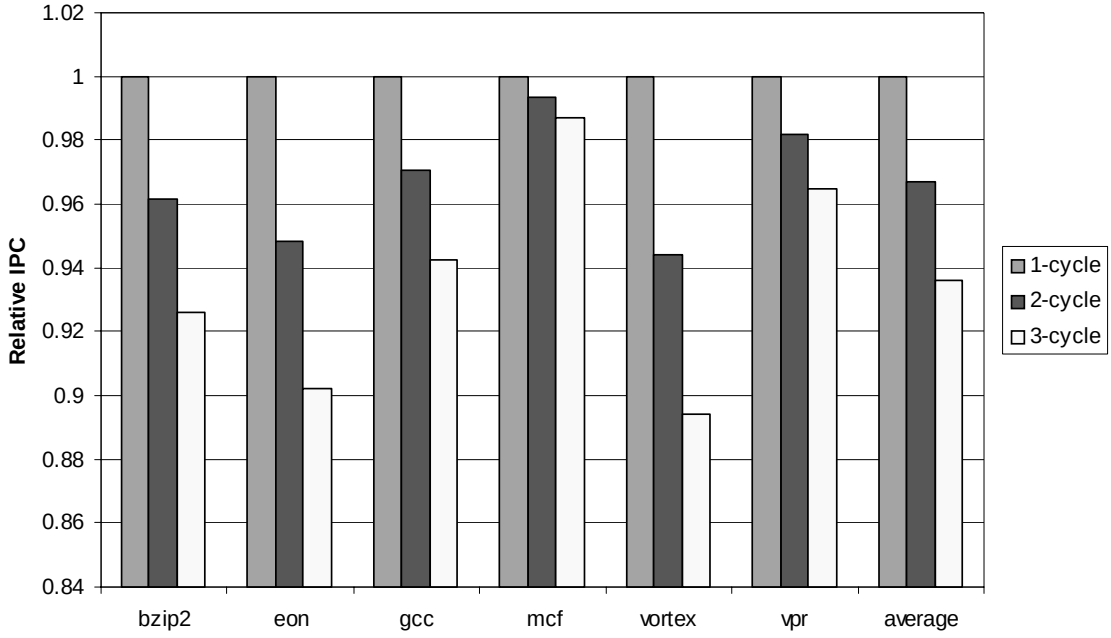


Figure 42. Relative IPC for various branch flush penalties with not taken branch prediction

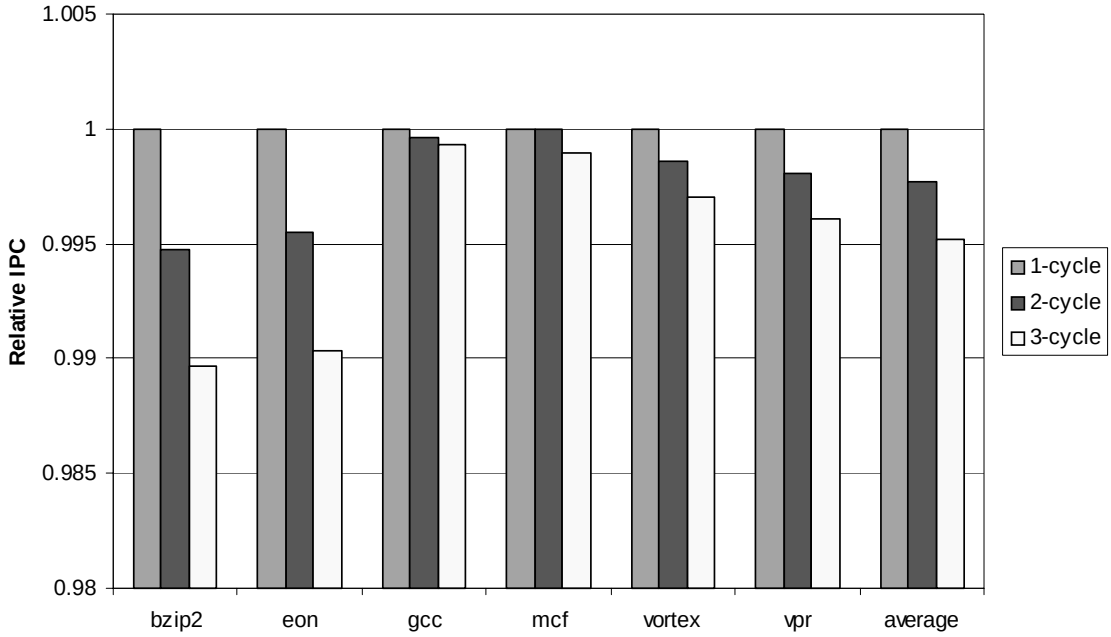


Figure 43. Relative IPC for various branch flush penalties with advanced branch prediction

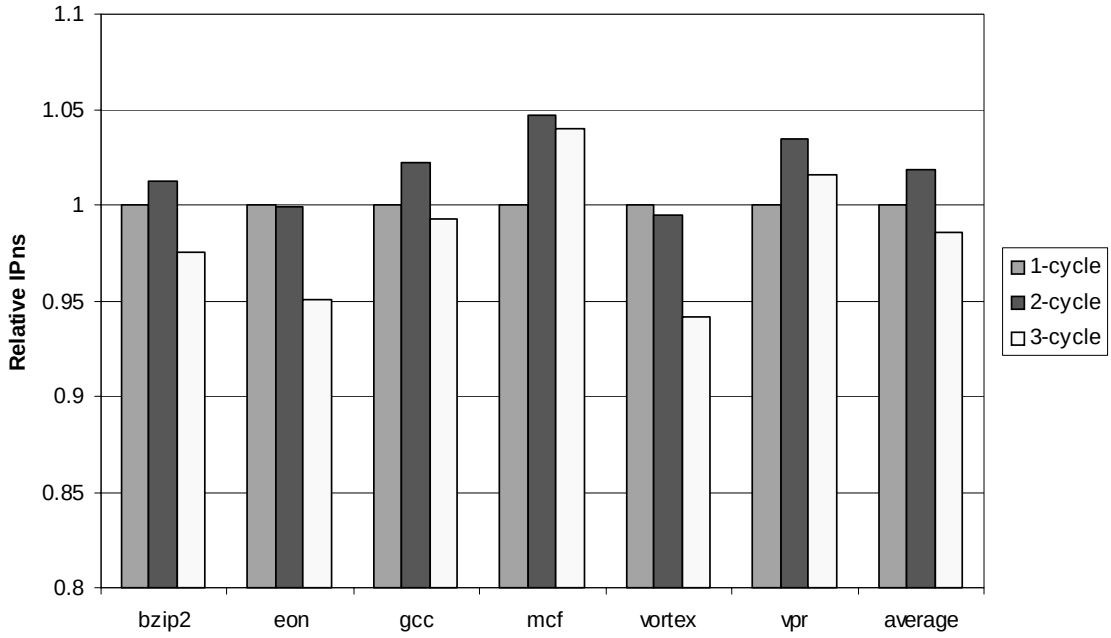


Figure 44. Relative IPns for various branch flush penalties with not taken branch prediction

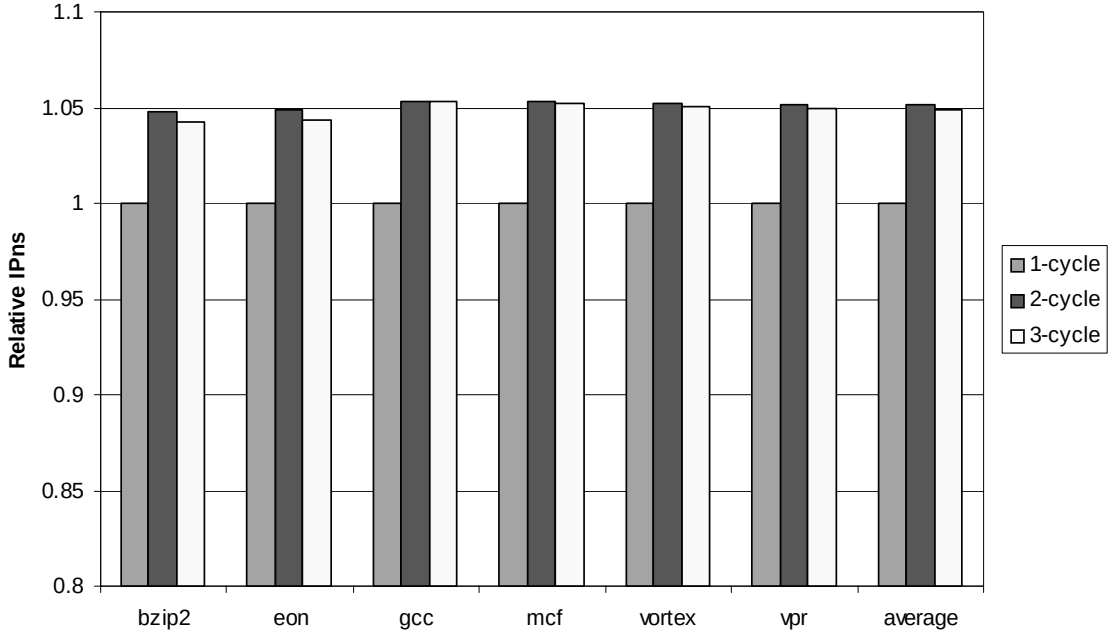


Figure 45. Relative IPns for various branch flush penalties with advanced branch prediction

Figure 44 and Figure 45 show the total processor throughput rate in instructions per nanosecond, using the 5.0% possible frequency boost from removing the branch signal from the critical path. With very poor branch prediction, and therefore high utilization of the misprediction signal, the 3-cycle counterflow flush design would lose ground in performance, while the 2-cycle design would roughly break even, depending on the running program. For the prototype chip with the addition of some advanced branch prediction, the possible increase in clock speed far outweighs the small IPC penalties, improving overall throughput by 4.4 - 4.7%. As this was the signal limiting the clock speed of the entire chip, the total throughput numbers are reflective of chip speedup, and not just of the complexity of the single signal.

5.3 Summary

As smaller chip features are fabricated, delays from long cross-stage signals will become prohibitive. Counterflow flush and micro rollback show promise as generalistic mechanisms for mitigating the circuit latencies associated with these control signals. As test cases, the two techniques were applied successfully to the Razor prototype chip, pipelining two long signals from the critical path, with very small impacts on throughput per cycle.

The advantage of this kind of pipelining approach is two-fold. First, the long, slow, power hungry wires are removed from the critical path. Second, it removes those elements while keeping the processor design logically very similar to a baseline processor. It would therefore be very simple to include in a design, and would avoid the problems introduced by changing the architecture drastically, as is done in projects such as TRIPS [56][64] or WaveScalar [75].

CHAPTER 6

CONCLUSION

6.1 Summary of the Thesis

There are three main contributions of this thesis: the tag elimination approach to improve efficiency in content-addressable memory schedulers, a competitive broadcast-free scheduler in the Cyclone, and a pair of methods to reduce complexity in long broadcast control signals. In addition, we studied the trade-offs that must be examined to determine the viability and performance of all of these techniques.

Our tag elimination approach was shown to reduce the cycle time and power consumption of a high-speed dynamic scheduler by up to 75%. Instead of increasing clock speed by attempting to break up the scheduling stage, as was done in some previous work, this technique attempts to improve the efficiency of the circuit, keeping it in a single stage. By removing traditionally over-designed comparison circuitry from the critical path, the loads on the tag broadcast signals were greatly reduced, with very little cost in instruction throughput per cycle. Further gains were demonstrated from reducing the scheduling process to react to only the most specific event - the readiness of its final input. While tag elimination can greatly reduce the complexity of dynamic scheduling, it is still, at its core,

a broadcast technique. Because of this, it is bound to the broadcast model of window scalability, growing quadratically in time and power with the desired size of an instruction window.

We introduced Cyclone as an alternative to the broadcast model of dynamic scheduling. Instead of sending result information to all waiting instructions, a more distributed peer-to-peer model of communication was adopted. The resulting structure was drastically smaller in area and faster than schedulers using the broadcast model, with respectable losses in instruction throughput per cycle. Further, the Cyclone scheduler included memory instructions as a primary element in determining latencies, which allows the schedule to include a more complete picture of execution.

Cyclone replaced a high-complexity structure with a more efficient option, while leaving most of the rest of the processor architecture the same as it was. To accomplish a similar goal, we present two simple methods for doing the same with long wires. Micro rollback and counterflow flush remove high-load signals from the critical path, replacing them with multi-cycle signals that do not change the overall operation of the rest of the processor.

As smaller chip features are fabricated, delays from long cross-stage signals will become prohibitive. Counterflow flush and micro rollback show promise as generalistic mechanisms for mitigating the circuit latencies associated with these control signals. The changes they make to the microarchitecture accomplish this task with minimal impact to the normal operation of the processor. As test cases, the two techniques were applied successfully to the Razor prototype chip, pipelining two long signals and removing them from the critical path, with very small impacts on throughput per cycle.

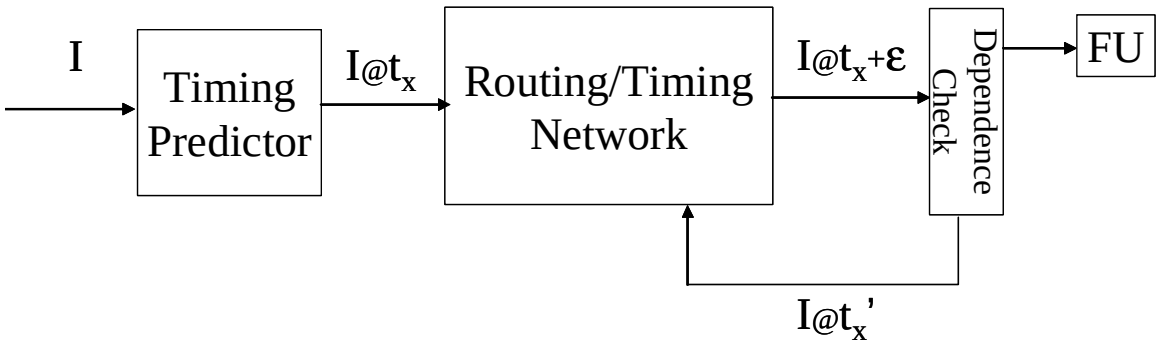


Figure 46. Broadcast-free dynamic scheduling element breakdown

6.2 Future Directions

There are several available avenues for future work on these topics. We have divided them into the categories of: improvements for broadcast-free dynamic scheduling and optimizations to the design-time process for reducing communication delays.

6.2.1 Broadcast-free Dynamic Scheduling

The Cyclone design is only one point in the broadcast-free scheduling design space. As is diagrammed conceptually in Figure 46, there are two major “black box” elements of the design which can be improved.

The timing network element, because it impacts the IPC drastically, is a prime candidate for improvement, as evidenced by several researchers [1][16][33] who published work in this area shortly after our original publication. Since this structure is conceptually very close to a routed network, its possible that ideas from fields such as network architecture and large memory system architecture could hold value in increasing the accuracy of the stage in constructing a predicted schedule.

The timing prediction element is also a candidate for optimization, as it was the clock cycle bottleneck for the Cyclone design. One possible direction of work is to examine how

much of the timing prediction could be off-loaded into a compiler, allowing the predictor unit to become simpler or even removed completely.

6.2.2 Improving the Control Signal Design Process

There are several interesting directions that can be explored in the streamlining of a design process that supports optimizations such as micro rollback and counterflow flush, and in finding new ways to use these trade-off designs.

The process we used in this work to examine trade-offs of circuits used a full, completed chip design to examine its bottleneck signals. In a real design, if decisions about using these pipelining techniques are put off until the end of a chip design cycle, any changes may create major problems. One possibly beneficial track of research would be to examine ways to identify problem signals much earlier in the process. This could be accomplished by using quantitative estimations on the required loading and length of each signal, possibly based on a coarse-grained weighting of signals in a first-run HDL netlist, along with signal usage data from architectural simulation.

Another possible research tack would be to make an attempt at automating portions of the design analysis. One approach to this would be allowing CAD-type tools to insert multi-cycle delays on signals meeting certain criteria. Another option would be to have architectural simulators flag logical signals that are not used often for possible optimization. We anticipate that the most difficult problem would be finding automated methods of combining the candidates from these two searches.

The focus of the optimizations in Chapter 5 is on increasing clock speeds. If the power consumed by wires is found to be very significant in a design, another possible application

of micro rollback and counterflow flushing would be to pipeline wires that are not on the critical path, solely for the purpose of reducing power consumption.

Finally, there may be other ways we can exploit a processor's ability to roll back. While micro rollback, by itself, cannot react to a processor's dynamic state as easily as a mechanism like Razor, there may be possibilities to use it to reduce some design-time complexities. For example, there may be a race condition which is fairly simple to detect, but very difficult to handle properly. Micro rollback could be used to break up this race condition when it is detected, removing the need for a designer to handle it.

BIBLIOGRAPHY

[1] Jaume Abella and Antonio González. “Low-Complexity Distributed Issue Queue,” *Proceedings of the 10th Annual International Symposium on High-Performance Computer Architecture*, Feb 2004.

[2] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. “Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures”, *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[3] Arm, Ltd. *ARM1022E Technical Reference Manual*, Ref: DDI0237A, December 10th, 2001.

[4] Arm, Ltd. *ARM1136 r0p2 Technical Reference Manual*, Ref: DDI 0211E, February 14th, 2003.

[5] T. Austin, “DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design,” *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, Nov. 1999.

[6] Todd Austin, Eric Larson, Dan Ernst. “SimpleScalar: an Infrastructure for Computer System Modeling,” *IEEE Computer*, Volume 35, Issue 2, Feb. 2002.

[7] The Berkeley Predictive Technology Models, http://www-device.eecs.berkeley.edu/~ptm/.

[8] Bradford M. Beckmann and David A. Wood. “TLC: Transmission Line Caches,” *Proceedings of the 36th International Symposium on Microarchitecture* (MICRO-36), December 2003.

[9] E. Borch, E. Tune, S. Manne and J. Emer. “Loose Loops Sink Chips”, *Proceedings of the 8th International Symposium on High Performance Computer Architecture* (HPCA-8), January 2002.

[10] D. Brooks, V. Tiwari, and M. Martonosi. “Wattch: A framework for architectural-level power analysis and optimizations”, *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[11] M. Brown, J. Stark, and Y. Patt, “Select-Free Instruction Scheduling Logic”, *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, December 2-5, 2001.

[12] D. Burger and T. M. Austin. "The SimpleScalar tool set, version 2.0", Tech. Rep. CS-1342, University of Wisconsin-Madison, June 1997.

[13] Ramon Canal and Antonio Gonzalez, “Reducing the Complexity of the Scheduling Logic”, *Proceedings of the 15th Annual International Conference on Supercomputing* (ICS-01), June 2001.

[14] George Z. Chrysos and Joel S. Emer. Memory Dependence Prediction Using Store Sets. *Proceedings of the 25th International Symposium on Computer Architecture* (ISCA-25), June 1998.

[15] Compaq Computer Corporation. *Alpha 21264 Microprocessor Hardware Reference Manual*, June 2001.

[16] Todd Ehrhart and Sanjay Patel. "Reducing the Scheduling Critical Cycle using Wakeup Prediction," *Proceedings of the 10th Annual International Symposium on High-Performance Computer Architecture*, Feb 2004.

[17] Dan Ernst and Todd Austin. "Efficient Dynamic Scheduling through Tag Elimination," *Proceedings of the 29th International Symposium on Computer Architecture* (ISCA-29), May 2002.

[18] Dan Ernst, Andrew Hamel, and Todd Austin. "Cyclone: A Broadcast-free Dynamic Instruction Scheduler with Selective Replay," *Proceedings of the 30th International Symposium on Computer Architecture* (ISCA-30), June 2003.

[19] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and Trevor Mudge. "Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation," *Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture* (MICRO-36), December 2003.

[20] J. A. Farrell and T. C. Fischer. "Issue logic for a 600-MHz Out-of-Order execution microprocessor," *IEEE Journal of Solid State Circuits*, 33(5), 1998.

[21] Daniele Folegnani and Antonio Gonzalez, "Energy-Effective Issue Logic", *Proceedings of the 28th Annual International Symposium on Computer Architecture*, June 2001.

[22] Antonio Gonzalez, Jose Gonzalez and Mateo Valero. "Virtual-Physical Registers", *Proceedings of the 4th Annual International Symposium on High-Performance Computer Architecture* (HPCA-4), Feb, 1998.

[23] Ricardo Gonzalez and Mark Horowitz. "Energy Dissipation in General Purpose Microprocessors", *IEEE Journal of Solid-State Circuits*, 31(9):1277-1284, September 1996.

[24] R. Gonzalez, B. Gordon, and M. Horowitz, "Supply and Threshold Voltage Scaling for Low Power CMOS," *IEEE Journal of Solid-State Circuits*, 32 (8), August 1997.

[25] Masahiro Goshima et al. "A High-Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors," *Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture* (MICRO-34), December 2001.

[26] A. Hartstein and T. Puzak. "The Optimum Pipeline Depth for a Microprocessor," *Proceedings of the 29th International Symposium on Computer Architecture* (ISCA-29), May 2002.

[27] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*, 3rd edition, Morgan Kaufmann Publishers, 2002.

[28] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, Patrice Roussel. “The Microarchitecture of the Pentium 4 Processor,” *Intel Technology Journal*, 1st quarter 2001.

[29] R. Ho, K. Mai, M. Horowitz. “Efficient On-Chip Global Interconnects,” *IEEE Symposium on VLSI Circuits*, June 2003.

[30] R. Ho, K. Mai, M. Horowitz. “Managing Wire Scaling: A Circuit Perspective,” *IEEE Interconnect Technology Conference*, June 2003.

[31] Mark Horowitz, Ron Ho, and Ken Mai. “The future of wires,” *Seminconductor Research Corporation Workshop on Interconnects for Systems on a Chip*, May 1999.

[32] M. Hrishikesh, N. Jouppi, K. Farkas, D. Burger, S. Keckler, and P. Shivakumar. “The Optimal Logic Depth per Pipeline Stage is 6 to 8 FO4 Inverter Delays,” *Proceedings of the 29th International Symposium on Computer Architecture* (ISCA-29), May 2002.

[33] Jie S. Hu, N. Vijaykrishnan and Mary Jane Irwin. “Exploring Wakeup-Free Instruction Scheduling,” *Proceedings of the 10th Annual International Symposium on High-Performance Computer Architecture*, Feb 2004.

[34] Intel Corporation. *Intel Itanium Architecture Reference Manual*, http://www.intel.com/design/itanium/manuals.htm.

[35] Keller, J. 1996. “The 21264: a superscalar Alpha processor with out-of-order execution”, Presented at the *9th Annual Microprocessor Forum*, San Jose, CA.

[36] Muhammad Khellah, James Tschanz, Yibin Ye, Siva Narendra and Vivek De. “Static Pulsed Bus for On-Chip Interconnects,” *Proceedings of the 2002 Symposium On VLSI Circuits*, June 2002.

[37] Ho-Seop Kim and James E. Smith. “An Instruction Set Architecture and Microarchitecture for Instruction Level Distributed Processing,” *Proceedings of the International Symposium on Computer Architecture* (ISCA-29), May 2002.

[38] Ilhyun Kim and Mikko Lipasti. “Half-Price Architecture,” *Proceedings of the 30th International Symposium on Computer Architecture* (ISCA-30), June 2003.

[39] G. Kucuk, K. Ghose, D. Ponomarev, and P. Kogge, “Energy-Efficient Instruction Dispatch Buffer Design for Superscalar Processors”, *Proceedings of the 2001 International Symposium on Low Power and Energy Design* (ISLPED ‘01), August 2001.

[40] Alvin R. Lebeck, Jinson Koppanalil, Tong Li, Jaidev Patwardhan, and Eric Rotenberg. “A Large, Fast Instruction Window for Tolerating Cache Misses,” *Proceedings of the International Symposium on Computer Architecture* (ISCA-29), May 2002.

[41] Seokwoo Lee, Shidhartha Das, Valeria Bertacco, Todd Austin, David Blaauw, and Trevor Mudge, “Circuit-Aware Architectural Simulation,” *Proceedings of the 41st Design Automation Conference* (DAC-2004), June 2004.

[42] N. Magen, et al. “Interconnect-Power Dissipation in a Microprocessor,” *Proceedings of the 2004 international workshop on System level interconnect prediction*, pages 7-13, Feb 14-15, 2004.

[43] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W.M. W. Hwu, B. R. Rau, and M. S. Schlansker. “Sentinel scheduling for VLIW and superscalar processors,” *ACM Transactions on Computer Systems*, 11(4):376--408, 1993.

[44] S. Manne, D. Grunwald, A. Klauser, "Pipeline Gating: Speculation Control for Energy Reduction", *Proceedings of the 25th Annual International Symposium on Computer Architechture*, June 1998.

[45] D. Matzke, “Will Physical Scalability Sabotage Performance Gains?” *IEEE Computer*, pages 37-39, September 1997.

[46] Scott McFarling. “Combining branch predictors”, Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[47] United States Patent #6,212,626, “Computer processor having a checker”, Amit A. Merchant and David J. Sager, assigned to Intel Corporation, issued April 3, 2001.

[48] P. Michaud, A. Seznec. “Data Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors”, *Proceedings of the 7th Annual International Symposium on High-Performance Computer Architecture* (HPCA-7). January 2001.

[49] Michael F. Miller, Kennneth J. Janik, and Shih-Lien Lu. “Non-stalling Counterflow Architecture,” *Proceedings of the Conference on High Performance Computer Architecture* (HPCA-4), May 1998.

[50] E. Morancho, J.M. Llaberia, A. Olive. “Recovery Mechanism for Latency Misprediction,” *Proceedings of the 2001 International Symposium on Parallel Architectures and Compilation Techniques* (PACT-2001), September 2001.

[51] The MOSIS Service, http://www.mosis.com

[52] Steven S. Muchnick. *Advanced Compiler Design & Implementation*, Morgan Kaufmann Publishers, 1997.

[53] T. Mudge. “Power: A first class design constraint,” *IEEE Computer*, vol. 34, no. 4, April 2001, pp. 52-57.

[54] J.M. Mulder, N.T. Quach, and M.J. Flynn. “An Area Model for On-chip Memories and its Application,” *IEEE Journal of Solid-State Circuits*, Volume 26 Issue 2, Feb 1991.

[55] S. Mutoh. “1V Power Supply HighSpeed Digital Circuit Technology with Multithreshold Voltage CMOS”. *International Journal of Solid-State Circuits*, 30(8):847–853, August 1995.

[56] R. Nagarajan, K. Sankaralingam, D. Burger, S. Keckler. “A Design Space Evaluation of Grid Processor Architectures,” *Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture* (MICRO-34), December 2001.

[57] Alexandru Nicolau. “Run-Time Disambiguation: Coming with Statically Unpredictable Dependencies,” *IEEE Transactions Computers*, Volume 38 No. 5, May 1989.

[58] Kevin J. Nowka, “High-Performance CMOS System Design using Wave Pipelining”, Stanford University Ph.D. Thesis, September 1995.

[59] Subbarao Palacharla, Norman P. Jouppi and J.E. Smith. "Complexity-Effective Superscalar Processors”, *Proceedings of the 24th Annual International Symposium on Computer Architecture*, May 1997.

[60] Subbarao Palacharla, Norman P. Jouppi and J.E. Smith. "Quantifying the Complexity of Superscalar Processors", Tech. Rep. CS-1328, University of Wisconsin-Madison, May 1997.

[61] S. Raasch, N. Binkert, and S. Reinhardt. “A Scalable Instruction Queue Design Using Dependence Chains,” *Proceedings of the 29th Annual International Symposium on Computer Architecture* (ISCA-29), May 2002.

[62] Glenn Reinman and Norm Jouppi, “An Integrated Cache Timing and Power Model”, Compaq Technical Report, http://www.research.compaq.com/wrl/people/jouppi/cacti2.pdf.

[63] Anne Rogers and Kai Li, “Software Support for Speculative Loads,” *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS ‘92), 1992.

[64] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. Keckler, C. Moore. “Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture,” *Proceedings of the 30th Annual International Symposium on Computer Architecture* (ISCA-30), June 2003.

[65] E. Schnarr and J. Larus. “Fast Out-Of-Order Processor Simulation Using Memoization”, *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 283--294, October 1998.

[66] A. Seznec, E. Toullec, O. Rochecouste. “Register Write Specialization Register Read Specialization: A Path to Complexity Effective Wide Issue Superscalar Processors,” *Proceedings of the 35th International Symposium on Microarchitecture* (MICRO-35), November 2002.

[67] Timothy Sherwood, Erez Perelman, Greg Hamerly and Brad Calder. “Automatically Characterizing Large Scale Program Behavior,” *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS 2002), October 2002.

[68] Michael D. Smith, Mark Horowitz and Monica S. Lam. “Efficient Superscalar Performance Through Boosting,” *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS ‘92), 1992.

[69] SPEC System Performance Evaluation Committee, www.spec.org.

[70] Eric Sprangle and Doug Carmean. “Increasing Processor Performance by Implementing Deeper Pipelines,” *Proceedings of the 29th International Symposium on Computer Architecture* (ISCA-29), May 2002.

[71] R. Sproull, I. Sutherland, and C. Molnar, “Counterflow Pipeline Processor Architecture,” Sun Microsystems Laboratories Inc. Technical Report SMLI-TR-94-25, April 1994.

[72] R.F. Sproull, I.E. Sutherland and C.E. Molnar. “The Counterflow Pipeline Processor Architecture,” *IEEE Design and Test of Computers*, Vol. 11 No. 3, Fall 1994.

[73] S. T. Srinivasan, A. R. Lebeck. "Load Latency Tolerance in Dynamically Scheduled Processors", *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, 148-159, 1998.

[74] J. Stark, M. Brown, and Y. Patt, “On Pipelining Dynamic Instruction Scheduling Logic”, *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, December 2000.

[75] Steve Swanson, Ken Michelson, Andrew Schwerin and Mark Oskin. “WaveScalar,” *Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture* (MICRO-36), December 2003.

[76] D. Sylvester and K. Keutzer, "Microarchitectures for systems on a chip in small process geometries," *Proceedings of the IEEE*, pp. 467-489, April 2001.

[77] D. Sylvester and K. Keutzer, "A global wiring paradigm for deep submicron design," *IEEE Transactions on CAD*, pp. 242-252, February 2000.

[78] D. Sylvester and K. Keutzer. “Rethinking deep-submicron circuit design,” *IEEE Computer*, 32(11):25–33, November 1999.

[79] D. Sylvester and K. Keutzer. “Getting to the Bottom of Deep Submicron II: A Global Wiring Paradigm,” *Proceedings of the International Symposium on Phyiscal Design* (ISPD-99), pages 193-200, 1999.

[80] Yuval Tamir, Marc Tremblay, and David A. Rennels, "The Implementation and Application of Micro Rollback in Fault-Tolerant VLSI Systems," *Proceedings of the 18th Fault-Tolerant Computing Symposium*, Tokyo, Japan, June 1988.

[81] M. Taylor, et al. “The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs,” *IEEE MICRO*, Mar/Apr 2002.

[82] Marc Tremblay and Yuval Tamir, "Fault-Tolerance for High-Performance Multi-Module VLSI Systems Using Micro Rollback," *Advanced Research in VLSI: Decennial Caltech Conference on VLSI*, Pasadena, CA, March 1989.

[83] Marc Tremblay and Yuval Tamir, "Support for Fault Tolerance in VLSI Processors," *International Symposium on Circuits and Systems*, Portland, OR, May 1989.

[84] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. “Exploiting choice: Instruction fetch and issue on an implementable

simultaneous multithreading processor", *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191-202, May 22-24, 1996.

[85] John F. Wakerly. *Digital Design Principles and Practices*, 3rd edition, Prentice Hall, 2001.

[86] E. Waingold, et al. "Baring it all to Software: Raw Machines," *IEEE Computer*, September 1997.

[87] K.C. Yeager. "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro*, Volume 16, Issue 2, April 1996.