The Dissertation Committee for Mary Douglass Brown
certifies that this is the approved version of the following dissertation:

Reducing Critical Path Execution Time by Breaking Critical Loops

Committee:

Yale Patt, Supervisor

Craig Chase

Donald S. Fussell

Stephen W. Keckler

Balaram Sinharoy

Reducing Critical Path Execution Time by Breaking Critical Loops

by

Mary Douglass Brown, B.S.; B.A.; M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2005

Acknowledgments

My years of graduate school would never have happened without the life-long support of my family. I want to thank my parents, Jack and Jane Brown, for being such excellent role models. Since I have been living in Austin, I have really enjoyed the comfort of living close to my sister and her husband, Terry and Jeff Stripling. Who knew we would both end up in Austin! I would also like to thank the Okies for providing such a wonderful environment to grow up in. After all, it takes a village to raise a child.

I have been part of a wonderful research group during the past seven years. I would like to thank my advisor, Yale Patt, for providing the right combination of guidance and freedom required to produce successful, independent-thinking Ph.D. graduates. Among the more senior members of the HPS research group, I would first like to thank Jared Stark, who first got me working in the area of dynamic scheduling. He has been an outstanding mentor and colleague over the years. I would also like to thank Sanjay Patel and Dan Friendly, who first got me working in the area of clustering during my second year at Michigan. Little did I know at the time I would come back to clustering as part of my dissertation. Marius Evers, who was my TA when I took EECS 470 at Michigan, as well as Robert Chappell and Paul Racunas, were always great sources of technical expertise. Rob also wrote the bulk of the simulator used to produce the performance results shown in this dissertation. I would also like to thank the other current members of the HPS research group: Sangwook Kim, Francis Tseng, Hyesoon Kim, Onur Mutlu, Moinuddin Qureshi, Dave Thompson, and Santhosh Srinath, for their friendship and contributions to the group's simulation infrastructure. Along with the newest members of the HPS research group, they have kept our research group flourishing at UT. It has been a pleasure working with them.

I would also like to thank Craig Chase, Don Fussell, Steve Keckler, and Balaram Sinharoy for taking the time to serve on my dissertation committee. They were all very pleasant to work with.

I have had several sources of financial support over the years. I would first like to thank the University of Michigan for the Rackham Engineering Fellowship, which is the reason I chose Michigan for graduate school. I would especially like to thank IBM who supported me on a Doctoral Fellowship for three years. I would also like to thank the University of Texas College of Engineering for providing a doctoral fellowship during my last two years at UT.

Finally I would like to thank Motorola, IBM, and Intel for providing me with summer jobs. These jobs have been crucial for developing my knowledge of microprocessor design.

Reducing Critical Path Execution Time by Breaking Critical Loops

Publication No. ___________________

Mary Douglass Brown, Ph.D.
The University of Texas at Austin, 2005

Supervisor: Yale Patt

Increasing bandwidth and decreasing latency are two orthogonal techniques for improving program performance. However, many studies have shown that microarchitectural designs that improve one of these may have a negative effect on the other. For example, bypass latency, or the time it takes to forward a result from one functional unit to another, may increase as the number of functional units increases because of longer wires and multiplexor delays. As another example, techniques used to exploit ILP such as out-of-order execution with large instruction windows will increase dynamic instruction scheduling latency. While exploiting ILP allows the number of instructions processed per cycle (IPC) to be increased, the increased scheduling latency may lower clock frequency, increase power consumption, or even negatively impact IPC if additional cycles are needed to schedule instructions for execution. This dissertation addresses microarchitectural techniques that allow ILP to be increased but have traditionally had a negative impact on the latency of the execution core.

Critical loops are pieces of logic that must be evaluated serially by dependent instructions. A program's critical path is the set of instructions that determine the execution time of a program, and this set of instructions is determined in part by critical loop latency.

The length of a program's critical path can be shortened by breaking critical data dependence loops, thereby improving performance. This dissertation introduces ways to break three critical data dependence loops: the execute-bypass loop, the steering loop, and the scheduling loop. The execute-bypass loop is reduced by means of clustering and steering. The steering loop is broken by introducing scalable steering algorithms that perform as well as the best previously published algorithms that did not scale with issue width. The scheduling loop is broken into multiple smaller loops, thus reducing the critical path through the scheduling logic.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Program execution time depends on both a processor's maximum execution bandwidth and execution latency. Because of the inherent Instruction-Level Parallelism (ILP) found in applications, superscalar processors can execute more than one instruction at a time thereby increasing execution bandwidth. By doing work in parallel, the total execution time of a program can be reduced. However, even after providing enough hardware resources to execute several instructions in parallel, program execution time is still limited by execution latency because of the inherent data and control flow dependences found in applications.

Increasing bandwidth and decreasing latency are conceptually two orthogonal techniques for improving program performance. However, microarchitectural techniques to improve one of these usually have a negative effect on the other. For example, bypass latency, or the time it takes to forward a result from one functional unit to another, may increase as the number of functional units increases because of both longer wire delays and longer multiplexer delays. Also, techniques used to exploit ILP such as out-of-order execution will increase dynamic instruction scheduling latency. While exploiting ILP allows the number of instructions retired per cycle (IPC) to be increased, the increased scheduling latency may either lower clock frequency or even negatively impact IPC if additional cycles are needed to schedule instructions for execution. This dissertation addresses microarchitectural techniques that are used to exploit ILP but have a negative impact on the latency of the execution core.

1.1 The Problem: The Limitation of Critical Paths

The critical path of a program is the set of instructions that limit the execution time of the program. The critical path is determined by the hardware design as well as the program's code and the program's inputs.

If a machine had infinite execution resources, the critical path could be determined simply by tracing the longest path through the data and control flow graph of a program. However, with limited hardware resources, the critical path is also affected by the size of the out-of-order window and contention for resources. For example, a load instruction that misses in the data cache has a long latency. Because instructions are retired from the instruction window in program-order, this long-latency instruction may block retirement of younger instructions that have finished execution. The fetch unit will continue to fill the instruction window until it is full. However, there may be instructions further down the instruction stream that cannot be put into the window that are independent of the load and would be able to execute if the window size were larger. Hence, even if the load has no data dependants, it can still affect the critical path of the program. Contention for resources such as functional units can also affect a program's critical path. An instruction's execution time may be delayed because of the unavailability of functional units or cache ports, thus putting it on the critical path.

By providing more hardware resources (i.e. bigger windows and more functional units), the critical path through the program can be shortened. However, these additional resources do not alter the data and control-flow graph of the program, which ultimately limit the length of the program's critical path. Ultimately, the length of the critical path is determined by critical loop latency. *Critical loops* are pieces of logic that must be evaluated serially by dependent instructions. Figure 1.1 shows a simple pipeline with four critical loops: the branch resolution loop, the steering loop, the scheduling loop, and the execute-bypass loop. The branch resolution loop consists of all of the pipeline stages from the stage

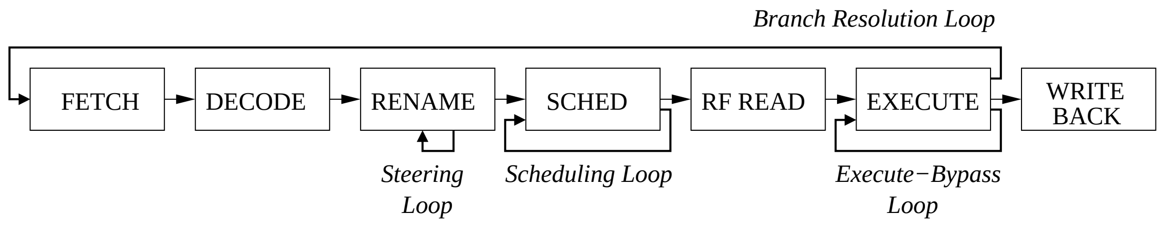


Figure 1.1: Common Critical Loops in a Processor Pipeline.

where branch direction and branch targets are predicted to the execute stage where branches are resolved. This loop is caused by control-flow dependences: all instructions after the branch are dependent upon the branch's resolution. With the use of branch prediction, the latency of the branch resolution loop is only incurred upon a misprediction. The steering loop is a small loop within the steering logic. The steering logic determines where to send instructions within the execution core after they have been renamed. With some steering algorithms, an instruction cannot be steered until all older instructions have been steered. Hence the steering logic may contain a small critical loop evaluated serially in program order by all instructions. A third loop is the dynamic instruction scheduling loop. Because an instruction cannot be scheduled for execution until the instructions producing its source operands have been scheduled, the scheduling logic forms a critical loop. Fourth, the execute-bypass loop consists of the logic required to execute an instruction and bypass its result to a dependent instruction. An instruction cannot execute until its source operands are produced, so the execute-bypass loop consists of both the structures needed to compute an instruction's result (e.g. an ALU or a data cache) as well as the bypass network that forwards results to dependent instructions.

As processor bandwidth is increased by providing more hardware resources, the impact of critical loop latency on performance becomes even larger. Figure 1.2 shows the IPC degradation on the SPEC2000 integer benchmarks when the scheduling, execution, and branch loops are increased by one cycle for a processor with four functional units.

The steering loop is only a fraction of one cycle, so it does not makes sense to stretch the loop over an additional cycle. Hence, results for stretching this loop over multiple cycles are not shown. In the baseline, the scheduling loop is one cycle and the execute-bypass loop is equal to the execution latency. The branch loop is considerably longer (23 cycles) which is why increasing this by only one cycle had a smaller effect on IPC. Figure 1.3 shows the IPC degradation of stretching these three loops when the execution width is increased to 16. All other machine parameters remain the same (and will be discussed in Chapter 3). The fact that the IPC degradation is bigger when more execution resources are provided demonstrates that critical loop latency will become even more important as processor execution resources are increased in the future. Furthermore, the latency of these critical loops increases as execution width is increased, thereby exacerbating the impact on performance.

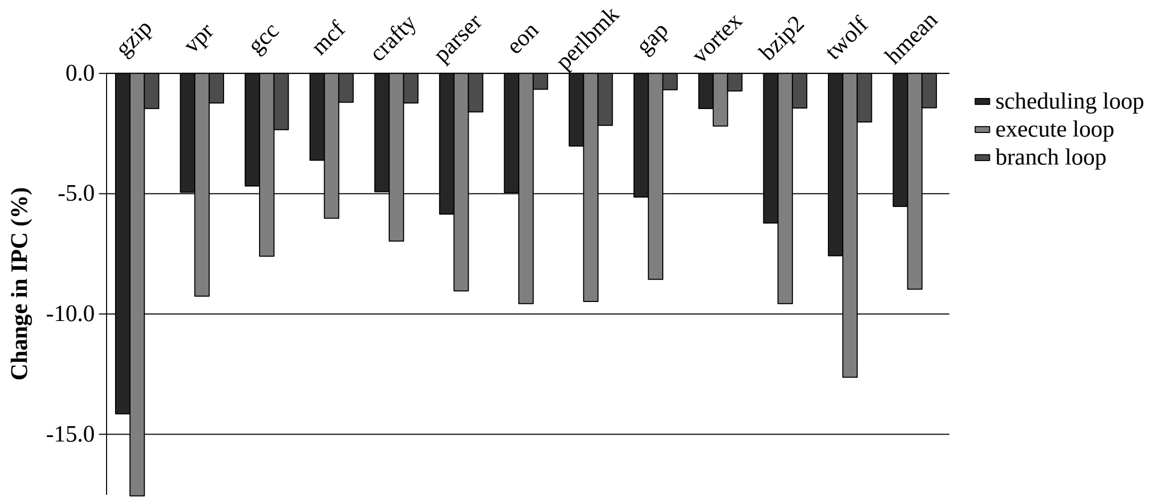


Figure 1.2: IPC degradation when increasing critical loops for 4-wide machine.

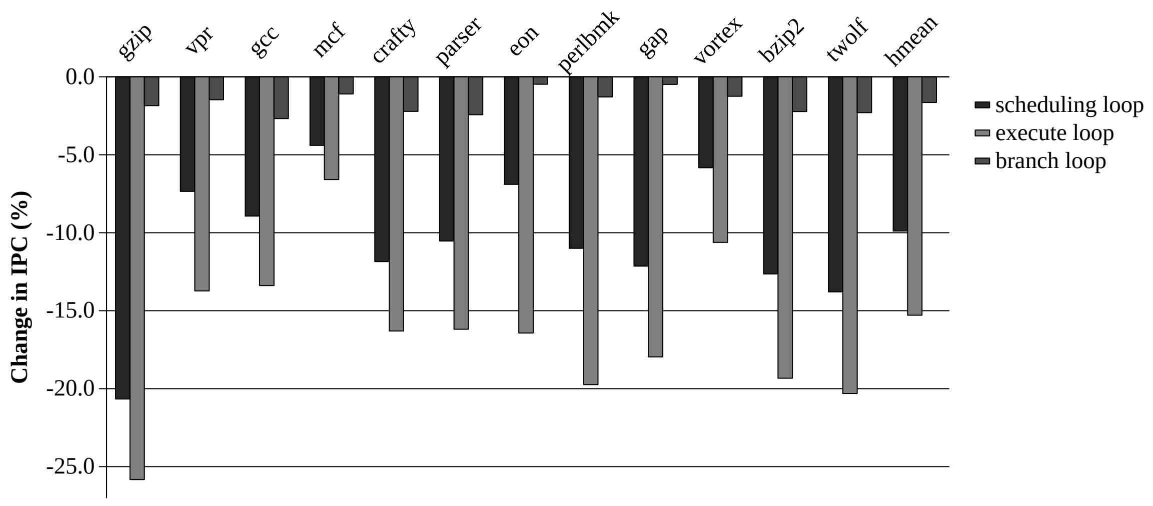


Figure 1.3: IPC degradation when increasing critical loops for 16-wide machine.

1.2 Critical Loops in the Execution Core

Once sufficient execution resources are provided, the performance bottlenecks are shifted to the critical loops. This dissertation focuses on those critical loops of the execution core whose latency increases as execution bandwidth increases, and it introduces techniques to break these critical loops.

1.2.1 Thesis Statement

The length of a program's critical path can be shortened by breaking critical data dependence loops, thereby improving performance.

1.2.2 Breaking Critical Loops

The execute-bypass loop is addressed by means of *clustering*. An execution core with many functional units may be partitioned into clusters to reduce the execute-bypass loop. An example of a clustered execution core is shown in Figure 1.4. In this exam-

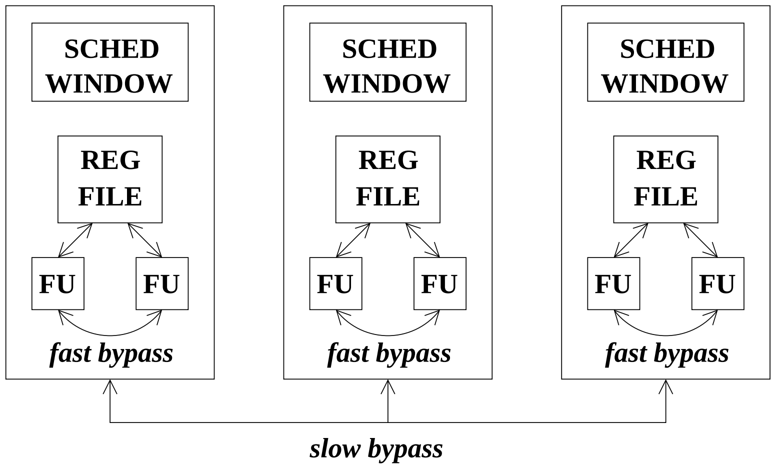


Figure 1.4: A Clustered Execution Core.

ple, parts of the scheduling window and physical register file, as well as a subset of the functional units, reside within each cluster. A result produced in one cluster can be quickly forwarded to any functional unit within the same cluster, while additional cycles are needed to forward the result to functional units in other clusters. By placing dependent instructions in the same cluster, most of the worst-case communication delays can be avoided, resulting in an overall performance improvement.

Steering is the process of assigning instructions to clusters. The steering logic is usually placed in conjunction with the rename logic since both operations require dependence information. It is desirable to steer an instruction to the same cluster as its *parent* (i.e. the instruction upon which it is dependent) to reduce forwarding latency. This means the instruction cannot be steered until after its parent has been steered. Hence the steering logic forms a critical loop. By breaking the critical steering loop, the steering logic can be parallelized.

Dynamic instruction scheduling can be described as a series of two steps: *wakeup*, and *select*. The wakeup logic determines when instructions are ready to execute, and the select logic picks ready instructions for execution. These operations are shown in Figure 1.5. When an instruction is ready for execution, it requests execution from the select logic. When it is selected for execution, it broadcasts a tag to notify dependent instructions that it has been scheduled. After a delay determined by its execution latency, the dependent instructions may wake up, thus completing the loop.[1] Because instructions cannot wake up until all of their parents have broadcast their tags, the wakeup and select logic form a critical loop.

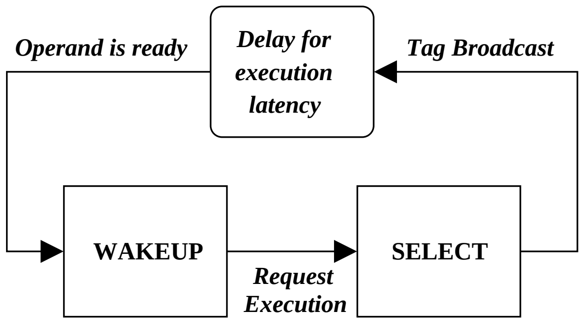


Figure 1.5: Logic in the Scheduling Loop

The critical scheduling loop can be broken by using *speculative scheduling*. Speculative scheduling techniques reduce the critical path of the logic forming the critical scheduling loop. The scheduling logic can then be pipelined over multiple cycles without requiring the scheduling loop to take multiple cycles.

Breaking critical loops can be advantageous in several ways. First, the out-of-order window size can be increased. The size of the instruction window is typically as large as it can be while still allowing dependent instructions to be scheduled in back to back cycles. By reducing the critical path of the scheduling loop, the number of entries in the window can be increased without affecting clock frequency. Second, clock frequency can

[1]In some implementations, the tag broadcast may occur after the delay.

be increased. By having less logic (or shorter wire delays) in a critical loop, the frequency can be increased without pipelining the loop over more cycles. Third, power can be reduced by using slower transistors in the logic that forms the critical loop.

1.3 Contributions

This dissertation introduces techniques for breaking three critical loops in the execution core: the execute-bypass loop, the steering loop, and the scheduling loop.

1. The length of the execute-bypass loop is reduced by means of clustering. This dissertation compares several clustering paradigms, and analyzes their performance and power. A mechanism called *Selective Cluster Receipt*, in which result broadcasts are only received by the clusters that need them, is introduced to save power for a given level of performance. The IPC, power, and scheduling and register file latencies of four paradigms are examined and compared: (1) a unified execution core, (2) a clustered execution core with a replicated register file, (3) a clustered execution core with a partitioned register file, and (4) a clustered execution core with a replicated register file implementing Selective Cluster Receipt. This dissertation demonstrates that for some clustering configurations it is better to replicate the register file across all clusters rather than partition it among clusters.

2. This dissertation provides insight as to why steering algorithms perform the way they do and exposes several problems that occur with existing steering algorithms. Better-performing steering algorithms that are scalable and can be used in high-bandwidth execution cores are introduced and evaluated. These scalable algorithms remove the critical loop that exists in the best previously proposed steering algorithms. Additionally, two steering mechanisms are introduced that improve performance: *out-of-order steering* and *virtual cluster assignments*. Out-of-order steering algorithms

both reduce the amount of inter-cluster communication penalties and reduce resource contention. Virtual Cluster Assignments significantly improve the performance of trace-cache based cluster assignment algorithms by improving load balancing.

3. The scheduling loop is broken into multiple smaller loops, thus reducing the critical path through the scheduling logic. Two speculative scheduling techniques, *Speculative Wakeup* and *Select-Free Scheduling*, are introduced and evaluated. With Speculative Wakeup, instructions predict when their parents' tags will be broadcast. The prediction occurs after their grandparents' tags have been broadcast. This technique allows the scheduling logic to be pipelined over two or more cycles. With Select-Free Scheduling, instructions assume they will be selected for execution as soon as they become ready to execute. This technique removes the select logic from the critical scheduling loop, allowing the wakeup logic to take one full cycle while still scheduling dependent instructions in back-to-back cycles.

1.4 Dissertation Organization

This dissertation is divided into eight chapters. Chapter 2 discusses related work in clustering, steering, and dynamic instruction scheduling. Chapter 3 describes the simulation infrastructure, including the performance and power models and benchmarks. Chapter 4 describes the general effects of clustering on IPC, scheduling window and register file latencies, and power. It also introduces Selective Cluster Receipt (SCR), and compares SCR to other clustered models. Chapter 5 discusses the impact of steering algorithms on performance and introduces several new algorithms. Chapter 6 discusses two techniques to reduce the critical scheduling loop: Speculative Wakeup and Select-Free Scheduling. Finally, Chapter 7 concludes, summarizes the experimental results which support the thesis statement, and provides suggestions for future work.

Chapter 2

Related Work

2.1 Clustering and Partitioning

In most clustered designs, the output of one functional unit can be forwarded to any other functional unit within the same cluster without incurring a cycle delay. This output is latched before being broadcast to other clusters, requiring one or more extra cycles to forward data between clusters. Hence dependent instructions may execute back-to-back (i.e. the consumer starts execution the cycle after the producer finishes) within the same cluster, but there is an inter-cluster forwarding penalty if the producer and consumer reside in different clusters. By reducing the minimum forwarding delay, the clock frequency of the execution core can be increased. However, IPC may be reduced because of the inter-cluster forwarding delays. The discussion in this section will be limited to only those microarchitectures which implement Sequential Architectures [58] (i.e. ISAs that don't explicitly specify dependences or independences between instructions) such as IA-32, PowerPC, or Alpha, but not VLIW or data-driven architectures.

This section will discuss several previously proposed clustered paradigms in which all clusters receive instructions from the same instruction stream. The following attributes are used to distinguish these paradigms:

- **which structures are replicated across clusters.** For example, do all clusters share the same physical register file, or does each cluster contain a local copy?

- **which structures are partitioned among clusters.** For example, if each cluster has a local physical register file, are all local register files mutually exclusive, identical, or do they share a subset of their registers?

- **how instructions are assigned to clusters.** The assignment may be made either dynamically using a trace cache or steering logic, or statically using a compiler, or a combination of the two.

- **the granularity of instruction distribution.** Is there a maximum or minimum limit on the number of consecutive instructions from an instruction stream assigned to the same cluster? Some paradigms may assign entire functions to the same cluster, while others have a limit on the number of instructions within a fetch packet that may be steered to the same cluster in a given cycle due to a restriction on the number of issue ports. The granularity of distribution is also determined by the inter-cluster communication penalty. A smaller granularity of distribution implies more inter-cluster communications because most instructions' register inputs were produced by the immediately preceding instructions from the instruction stream.

2.1.1 Multicluster Architecture

In the Multicluster Architecture [23], the physical register file, scheduling window, and functional units are partitioned into clusters, so the execution core can be run at a higher clock frequency compared to a processor with a centralized core. In this paradigm, each cluster is assigned a subset of the architectural registers, so the cluster assignment is performed statically by the compiler. If an instruction's register operands must be read from or written to more than one cluster, copies of the instruction must be inserted into more than one cluster. The primary problem with this paradigm is that it lowers IPC for several reasons. First, these copy instructions must contend with regular instructions for scheduling window write ports, register file ports, execution cycles, and space within the instruction window. Second, because the compiler does not have knowledge of how balanced the workload is over all clusters at runtime, it cannot evenly distribute the workload over all clusters. These two problems cause a degradation in IPC. The clustering paradigms used in this dissertation have fewer or no copy instructions, and the cluster assignment algorithms are better at load balancing.

2.1.2 Partitioned Structures with Dynamic Steering and Copy Instructions

In the clustered architecture described by Canal, Parcerisa, and González [16, 52], each cluster contains a partition of the physical register file and scheduling window, as well as a subset of the functional units. While dependent instructions within the same cluster can execute in back-to-back cycles, inter-cluster forwarding takes two or more cycles. An instruction's register result is only written to the partition of the physical register file that is in the instruction's local cluster. If an instruction needs a source operand that resides in a remote cluster, a special *copy* instruction must be inserted into the remote cluster. Only copy instructions may forward register values between clusters. By limiting the number of copy instructions that can be executed in a cycle, the number of register file write ports and global bypass paths can be reduced. This will reduce the register file and scheduling window access times, and potentially increase the clock frequency. Furthermore, since the entire physical register file is not replicated across clusters, each partition can have fewer entries than if the entire register file were replicated, which further reduces the register file access time. The reduction in the register file and scheduling window area can also save power in the execution core.

The primary difference between this paradigm and the Multicluster paradigm is that steering is performed dynamically, so any architectural register could be produced in any cluster. However, as with the register copies in the Multicluster paradigm, the copy instructions may lower IPC because they occupy resources and increase the length of the critical path of the data dependence graph. This paradigm will be evaluated in Chapter 4, and compared against alternative clustering paradigms. Results show that this paradigm still has a significant loss in IPC (without power savings) as compared to the alternative paradigms.

2.1.2.1 Consumer-Requested Forwarding

Hrishikesh [36] also uses a clustered model with copy instructions, but introduces a technique called *Consumer-Requested Forwarding* to reduce the number of copy instructions. Each cluster contains a table, called the IFB Table, which holds “inter-cluster forwarding bits”. These bits indicate in which clusters a given physical register result is needed. There is one entry in the table for each physical register stored in the local cluster, and each entry has one bit associated with each other cluster. A bit of that entry is set if there is a dependent instruction residing in another cluster.

The primary problem with this technique is that the table that indicates if and where a result is needed (the IFB) is located in the producing instruction’s cluster, not the consuming instructions’ clusters. The reason this is a problem is that there is a delay between the time that the dependence analysis logic detects a bit of the IFB table should be set and the time it actually gets set. Instructions may execute and read the IFB during this time, which presents the possibility of starvation. Hrishikesh [36] provides several solutions to this deadlock problem, however. Even with these solutions, simulation results showed that IPC was lowered if this delay, called the *detect-to-set delay*, was over three cycles. In a deeply pipelined machine that has several pipeline stages between rename and the last scheduling stage in addition to several cycles of inter-cluster forwarding delay, the detect-to-set delay may be considerably longer than three cycles.

A technique described in Chapter 4 to reduce inter-cluster forwarding and register file writes, called Selective Cluster Receipt, is not affected (in terms of IPC) by the delay between the rename and scheduling stages. With Selective Cluster Receipt, the information that indicates where a register value is needed is located in the consumers' clusters, not the producer's cluster. This requires all instructions to broadcast their tags and data along the *inter-cluster* portion of the bypass network, but this portion of the network is small compared to the *intra-cluster* portions of the bypass network.

2.1.3 Partitioned Structures with Dynamic Steering but no Copy Instructions

The clustered architecture described by Zyuban and Kogge [73] also uses a partitioned physical register file. However, rather than using copy instructions to broadcast results, dedicated hardware is used to support copy operations. Each cluster has an extra scheduling window for determining when values are ready to be copied to another cluster. This window is similar to a traditional scheduling window except that it may be smaller, and only one source is needed per operation, while real instructions may have two (or more) operands. Each cluster also has a CAM to hold tags and results that were broadcast from other clusters, rather than using extra physical register file entries as was done in the architecture described by Canal, Parcerisa, and González. There are two problems with this dedicated hardware approach. First, it assumes that an instruction can wake up, arbitrate for execution, and broadcast its tag to a copy operation which then wakes up, arbitrates for execution, and broadcasts its tag to a dependent instruction all within one clock cycle. In other words, there are two critical scheduling loops within one cycle, rather than just one, as in the other clustered paradigms. Hence it may not be able to run at as high a clock frequency as other paradigms. The second problem with this paradigm is that extra buffers are used to hold copy instructions, rather than just adding a few entries to each scheduling window. This adds additional complexity to the design. The clustered architectures used in this dissertation do not use dedicated structures to hold inter-cluster broadcasts and do not have two scheduling loops within one cycle.

2.1.4 Register Read and Write Specialization

The paper by Seznec et al. [61] proposes a paradigm with four symmetrical clusters in which the physical register file is broken into four subsets. With Read Specialization, each cluster has the capability to read from three of the four subsets: the local subset and the two most adjacent subsets. Read Specialization puts a constraint on the steering logic because instructions with source operands can only be steered to a subset of the four clusters and in some cases only one cluster. Read specialization suffers from load balancing problems, so it will not be evaluated in this dissertation.

With Write Specialization, instructions in a given cluster may write to only one of the four subsets. This reduces the number of write word lines in each subset. This technique is used to bank the register file used in the baseline processor register files in this dissertation, and is described in more detail in Section 3.1.5.

2.1.5 Paradigms with Replicated Register Files

The Alpha 21264 [29] is an example of a product with a clustered execution core. The functional units are partitioned into two clusters, and there is a 1-cycle penalty for forwarding results between clusters. Both clusters contain identical copies of the physical register file. When an instruction broadcasts its result, the result is written into both copies of the register file. A replicated register file will be used in the baseline clustered architecture in this dissertation.

The paper by Palacharla et al. [51] proposes a clustered microarchitecture in which out-of-order scheduling windows are replaced with in-order queues. Each cluster contains a set of FIFOs from which instructions are selected for execution. Each FIFO contains a chain of dependent instructions, and only instructions at the heads of FIFOs may be selected for execution. By reducing the number of instructions that could be selected for execution each cycle, the scheduling logic can be simplified. The main drawback to this technique is that it is difficult to balance the workload across FIFOs.

Another clustering paradigm with a centralized register file is the PEWs [40] (Parallel Execution Windows) microarchitecture. A PEWs processor contains several execution windows that are connected in a ring. Buffers between each adjacent pew hold register values that need to be broadcast to other pews. Only one value can be forwarded between adjacent pews per cycle. The buffers add latency to the inter-cluster communication, which makes this paradigm undesirable for the same reason as the other clustered paradigms requiring buffers for inter-cluster communication.

2.1.6 Multiscalar Processors

In the Multiscalar processing paradigm [63], a program's instruction stream is divided into tasks which are executed concurrently on several processing units. Each processing unit contains its own physical register file. Because there may be data dependences between the tasks, the processor must support register result forwarding and memory dependence detection between the processing units. Because each task is a contiguous portion of a program's dynamic instruction stream, only the live-out register values from a task must be forwarded to successive tasks executing on other processing units. The compiler can identify the instructions that may produce live-out register values, and inserts instructions called *release instructions* into the code indicating that the values may be forwarded to other processing units. This paradigm may be appropriate for programs that can be broken into tasks that communicate few values, but it still has some problems. First, the cost of communication between processing units is high for both register and memory values. Hence this paradigm is not as good as other clustering paradigms at exploiting fine-grained ILP. Furthermore, the compiler's job of breaking a program into appropriately-sized tasks has been shown to be quite difficult, and achieving high IPC with this paradigm is limited by control flow and data dependence predictions [67].

2.2 Other Approaches to Breaking the Critical Execute-Bypass Loop

In addition to clustering, several other approaches have been used to break the execute-bypass loop. One technique is to use pipelined digit-serial adders [20] to forward intermediate arithmetic results to subsequent instructions, as was done in the Intel Pentium 4 [35]. In this processor, the ALUs are *double-pumped*, meaning they are pipelined over two cycles while still allowing dependants to execute in back-to-back cycles. This is accomplished by computing the result for the lower two bytes in the first cycle and the upper two bytes in the second cycle. The carry-out of the 16th bit is passed from the first stage to the second stage. Redundant binary arithmetic [1] can also be used to pipeline the functional units while still allowing a dependence chain to execute in back-to-back cycles [8, 31]. All of these approaches are orthogonal to clustered paradigms, so clustered paradigms may use pipelined functional units that produce values in intermediate formats to further reduce the length of the execute-bypass loop.

2.3 Steering

Steering is the process of deciding to which cluster an instruction should be issued. Existing steering algorithms can be divided into three general classes: (1) static, (2) history-based, and (3) front-end. Static steering algorithms are performed by the compiler. Cluster assignments are indicated either by the architectural destination register number or the position of the instruction, in the case of a VLIW architecture. History-based algorithms are typically performed after instructions have executed and cluster assignments are stored in a table or trace cache. Front-end algorithms are performed after dependence analysis but before issuing instructions into the scheduling window. This section discusses previously proposed steering algorithms according to their classification.

2.3.1 Static Steering Algorithms

With *static* steering algorithms, cluster assignments are determined by the compiler. This is generally the case for some VLIW processors where the location of an instruction within a word determines its cluster assignment [27]. The paper by Sastry et al. [60] studies compile-time cluster assignment for a processor that had two clusters: one which executed only integer instructions, and one which executed both floating point and integer instructions. The Multicluster Architecture [23] also uses the compiler to assign instructions to clusters by assigning each cluster a subset of the architectural registers. Because static steering heuristics have no knowledge of dynamic microarchitecture state, they suffer from load balancing problems. Furthermore, because an instruction's architectural destination register determines its cluster assignment, all dynamic instances of the instruction are assigned to the same cluster. Poor load balancing lowers IPC.

2.3.2 History-Based Steering Algorithms

With *history-based* algorithms, cluster assignment is determined dynamically, and the cluster assignment is typically stored in a history table or as part of an instruction cache or trace cache. The cluster assignment is usually based on behavior of previous instances of that static instruction. In the work on fill-unit optimization by Friendly et al. [28], cluster assignments are stored in the trace cache. The fill unit analyzes the instruction stream at retire time and forms optimized traces to be stored in the trace cache. One optimization is to assign instructions to one of four clusters, each of which has four issue ports. The assignment algorithm works as follows: for each issue port, pick the first instruction in the trace that has source operands previously assigned to that port's cluster. If no instruction is found, then the first unplaced instruction is assigned to that port. One problem with history-based algorithms is that the cluster assignment logic does not have precise knowledge of all register dependences – only those dependences on earlier instructions within the same trace packet are known to be correct. Dependences on instructions in other trace packets can only be predicted.

The paper by Canal et al. [16] investigates steering algorithms for the two-cluster processor first introduced by Sastry et al. [60]. This processor has an integer cluster and an integer/floating-point cluster. The steering algorithms in this paper are based on the examination of the register dependency graph (RDG). They partitioned the instruction stream into *slices* and assign all instructions in the same slice to the same cluster. A *slice* is the set of all ancestors (i.e. producers, producers' producers, etc.) of a given instruction. The algorithms described in this paper use slices of either load or branch instructions. Slices are identified using a table of slice IDs indexed by PC. Instructions can update their parent's slice IDs upon execution. The steering paper by Baniasadi and Moshovos [3] shows that the slice algorithm does not perform as well as other steering algorithms discussed in this section.

More recently, Bhargava and John [4] have come up with a trace-cache based heuristic called Feedback-Directed Retire-Time (FDRT) assignment that shows further improvement in IPC over previous trace-cache algorithms. This algorithm distinguishes between "inter-trace" register dependences and "intra-trace" dependences. As in the paper by Friendly et al. [28], the fill unit is used to assign instructions to clusters. However, in [4], some instructions may be "pinned" to clusters. Once an instruction becomes pinned, it must keep the same cluster assignment for the entire time that its trace cache line is resident in the trace cache. The process of pinning prevents completely new cluster assignments for each trace cache line from being made every time the trace is retired and re-analyzed by the fill-unit. By adding this stability to the cluster assignments, instructions are more likely to be placed in the same cluster as their inter-trace dependents.

The FDRT algorithm works as follows: entries for each instruction stored in the trace cache can be designated as a *leader*, or a *follower*, or neither. Once an instruction is marked as a leader or follower, it stays that way for the duration of the lifetime of the trace cache line. An instruction is marked as a leader if it is not already a leader or follower and it has dependants which belong to younger fetch packets. An instruction is marked as a follower if the following hold true: (1) it has not already been marked as a leader or follower, (2) the instruction that produces its last-arriving source operand was part of a different fetch packet, and (3) that parent was marked as a leader or follower. When an instruction gets marked as a leader or follower, it is pinned to a particular cluster (the cluster of its parent that was pinned) until its trace cache line gets kicked out of the trace cache. Instructions that are not pinned are steered towards the cluster of their source operands, preferably towards source operands produced by instructions in a different fetch packet. This algorithm is implemented as a baseline history-based algorithm and discussed further in Section 5.1.1.4.

Another type of history-based steering uses critical path prediction. The work by Fields et al. [25] uses a critical path predictor to steer instructions to either slow or fast scheduling pipelines. Instructions that are latency-critical are steered to the fast pipeline, while others are steered to the slower pipeline.

History-based algorithms such as FDRT have the advantage that they do not increase the pipeline depth because the fill-unit is on the back end of the pipeline. The main problem with history-based algorithms is that they suffer from poor load balancing for some cluster configurations. This in turn limits their performance potential. Results of simulations revealing the load balancing problems in history-based algorithms are shown in Section 5.1, and a technique to improve the load balancing in history-based algorithms is described in Section 5.2.1.

2.3.3 Front-End Steering Algorithms

With *front-end* algorithms, cluster assignment is determined dynamically either during or after the instruction fetch stage. Unlike static and history-based algorithms, front-end algorithms may use knowledge of the current microarchitecture state at the time the instructions to be steered are about to be placed in the instruction window. However, they may not have knowledge (or a good prediction) of when and where an instruction's register result will be used by dependent instructions.

A dependence-based steering algorithm was used by Palacharla et al. [51] to assign instructions to one of several FIFOs in the execution core. When using a dependence-based algorithm, an instruction is usually assigned to the same cluster as its parents. There are several variations of this algorithm. Many variations use secondary heuristics to determine where to send an instruction. Examples of such heuristics are (1) how are instructions steered if their parents have already retired? (2) How are instructions steered if their parents have not retired, but have finished execution? (3) How are instructions steered if the desired cluster is already full? (4) Are other metrics used in the cluster assignment, such as the age of the parent instructions, predictions of the last available source operand, or the number of instructions waiting to execute in each cluster? Variations of dependence-based algorithms and improvements to them are evaluated in Chapter 5.

The steering paper by Baniasadi and Moshovos [3] includes a comparison of several classes of steering algorithms on a four-cluster processor with two functional units per cluster. They evaluated both *adaptive* and *non-adaptive* steering algorithms. The heuristics used by non-adaptive algorithms do not change over run-time, while the heuristics used by the adaptive algorithms may change. Chapter 5 evaluates two fundamental types of algorithms evaluated in their paper: modulo-N (MOD-N) and dependence-based (DEP) algorithms.

With the modulo algorithm, *n* consecutive instructions are assigned to each cluster before moving on to the next one. For example, when using MOD-6, the first six instructions would be assigned to cluster 0, the next six to cluster 1, and so on. For a given modulo value, a bound can be placed on the number of instructions that can be steered to each cluster in any given cycle. For example, when using a four-cluster processor with an issue width of 16, six ports are needed for the issue window in each cluster when using MOD-6. To implement an exact MOD-3 heuristic, six ports would be needed: two clusters would be allocated six instructions each, and the remaining two clusters would be allocated three instructions each, assuming 16 instructions total were issued.

The adaptive steering algorithms investigated by Baniasadi and Moshovos include voting-based methods and an adaptive-modulo algorithm. The voting-based methods used a PC-indexed lookup table with four 2-bit saturating counters per entry. The counters measured how much functional unit contention the instruction had experienced in the past. Instructions were steered to the cluster with the highest counter value (the cluster in which resource contention was least likely). If there was a tie among clusters, a backup non-adaptive algorithm was used to assign a cluster. With the adaptive-modulo algorithm, the modulo value could be changed over the course of the program. The best value was selected by periodically incrementing or decrementing the modulo value, and comparing how often instructions had to stall their execution when using each modulo value. Hence a local minimum could be found. Modulo steering algorithms do not perform as well as dependence-based algorithms because they do not consider data dependences.

The DCOUNT algorithm was introduced by Parcerisa, and González [15, 52]. This is a dependence-based algorithm which uses a load-balancing heuristic. The DCOUNT is a measure of load imbalance. If this imbalance reaches a threshold, then instructions will be steered to the least-loaded cluster. Otherwise, they are steered to a cluster using a dependence-based algorithm.

The DCOUNT algorithm is the best performing previously published steering algorithm. The problem with it along with other dependence-based algorithms is that it is not scalable to high-bandwidth front-ends (whether "high-bandwidth" means wide-issue or high-frequency). The reason why dependence-based algorithms are not scalable is discussed in Section 5.1.3. A solution that makes dependence-based algorithms more scalable is discussed in Section 5.2.2.

2.4 Scheduling

The schedulers introduced in this dissertation are designed to break the critical scheduling loop by pipelining the scheduling logic over multiple cycles. This section discusses other techniques that can be used to pipeline the scheduling logic. It also discusses other techniques that don't necessarily break the critical scheduling loop, but do potentially reduce scheduling latency.

Clustering is actually a form of pipelined scheduling because tag broadcasts are pipelined across clusters. Likewise, any scheduler with a partitioned window that takes multiple cycles to forward tags between partitions is pipelined. Section 2.4.1 describes proposals of partitioned schedulers.

Several schedulers have used prescheduling to reduce or eliminate the wakeup portion of instruction scheduling. These techniques are discussed in Section 2.4.2. Section 2.4.3 describes another speculative scheduler that removes the wakeup logic from the scheduling window. Other schedulers that reduce or eliminate tag broadcasts by using reverse data dependence pointers are described in Section 2.4.4. Finally, Section 2.4.5 describes other dynamic scheduling logic circuit designs.

2.4.1 Partitioning Schemes

The paper by Hrishikesh et al. [37] describes techniques to pipeline both the wakeup logic and the select logic. The wakeup logic is pipelined by means of partitioning the window and pipelining the tag broadcast across the partitions. The select logic is also pipelined by means of partitioning. Instructions from each partition of the window are first *pre-selected*, and then a subset of the pre-selected instructions are actually selected for execution. However, instructions in the highest-priority partition (containing the oldest instructions) do not have to be pre-selected. The request logic for instructions from that partition are not pipelined, but are fed directly to the final selection stage so that dependent instructions may execute in back-to-back cycles.

The scheduler proposed by Lebeck et al. [42] exploits the fact that most instructions that reside in the window for a long period of time are waiting on a load miss. With their technique, instructions that would normally wait in the scheduling window for a long period of time are moved to another structure called the Waiting Instruction Buffer (WIB). When the load miss is filled, the dependent instructions are then re-inserted into the scheduling window. While the implementation of the scheduling window is not modified by this technique, it does allow the effective size of the scheduling window to be increased without physically increasing its size.

The scheduler proposed by Brekelbaum et al. [6] divides the scheduling window into two partitions: a small, fast partition that holds latency critical instructions, and a large, slow window that holds latency tolerant instructions. All instructions are first issued into the slow window. A heuristic is used to select possibly latency-critical instructions to be moved to the fast window. The heuristic selects old, non-ready instructions to be moved. This scheduler is effective at exploiting far-flung ILP with the slow window while scheduling latency-critical instructions back-to-back in the small window.

All of these partitioning schemes share the same problem: the critical wakeup-select-broadcast loop must still be evaluated in a single cycle for a subset of instructions in the window or instructions in a particular partition of the window. This is not the case for the speculative scheduling techniques described in Chapter 6. With speculative scheduling, the critical scheduling loop can be pipelined while still allowing back-to-back execution of dependent instructions regardless of where the instructions are located within the window. The speculative scheduling techniques can be used in conjunction with the partitioning schemes described here.

2.4.2 Avoiding Wakeup by Prescheduling

Many proposed schedulers take advantage of the fact that most instruction latencies, except for loads, are known at decode time. These schedulers use a technique called *prescheduling*, in which estimated wakeup times are computed when instructions are in the decode stage. Canal and González [13] published the first such technique. When instructions are decoded, they check a table indexed by physical register, called the Distance Table, that stores the estimated time that register will be produced. Loads, as well as instructions dependent on loads, have invalid entries in the table until their availability time is known. Instructions that do not know their wakeup time when they are dispatched are placed in a Wait Queue which is similar to a scheduling window. Instructions that do know their wakeup time are placed in an in-order queue, and are positioned according to their wakeup time. When a load's availability time is known, it broadcasts its tag, along with its availability time, to the Wait Queue and the Distance Table. Dependent instructions in the Wait Queue can then then broadcast their tags to the Wait Queue, update the Distance Table, and move to the in-order queue. This technique is modified in their follow-on paper [14]. In this paper, loads are assumed to hit in the cache, so the Wait Queue is not needed. Instead, all instructions are first placed in the in-order queue. If there is a load miss, dependants are placed in a separate out-of-order scheduling window called the Delayed Issue Queue. Instructions may be scheduled out of either the Wait Queue or the Delayed Issue Queue in this technique.

A paper by Michaud and Seznec [44] also proposed prescheduling instructions before placing them in an in-order queue. The paper explains the prescheduler implementation in detail. Load-store dependences are predicted using a store-set predictor [18], although the simulations assume a perfect data cache.

Ernst et al. [22] proposed the Cyclone scheduler, which uses a prescheduler to place instructions in a countdown queue from which instructions are scheduled in-order. Instructions are always inserted into the tail of the countdown queue, but they may skip over a section of the queue using bypasses that are internal to the queue. The queue is actually circular; when instructions are at the head of the queue, they check a scoreboard to see if they are ready. If they are not (due to an earlier load latency misprediction), then they are re-inserted into the queue.

In follow-on work to the Cyclone scheduler, Hu et al. [38] developed techniques to avoid some of the problems with resource contention found in the Cyclone scheduler. They added selection logic that can select any instruction that is ready for execution from the queue. By removing instructions that are not at the head of the queue, there is less contention for issue queue slots in the latter half of the queue.

The problem with all prescheduling techniques is that they significantly increase pipeline depth. Instruction execution times must be computed before inserting instructions into the in-order queue. Because it is possible that a group of consecutive instructions form a serial dependence chain, this technique adds a serialization bottleneck to the front end of the pipeline. Ernst et al. describes a way to get rid of this bottleneck, but their solution is not scalable to wide-issue pipelines.

2.4.3 Other Speculative Schedulers

Ehrhart and Patel [19] proposed a scheduler which removes the wakeup logic by predicting the wakeup time of an instruction. When an instruction is issued, its wakeup time is calculated with two components: a predicted wakeup time and an allowance. An instruction's predicted wakeup time is a function of the static instruction's previous wakeup times and the estimated cost of replaying the instruction if there is a wakeup time misprediction. The allowance is added to the predicted wakeup time to re-adjust the wakeup time if replays occur while the instruction is waiting to execute. This technique suffers a 7% loss in IPC in an 8-wide machine with a 64-entry scheduling window and 2-cycle ALU latency. The speculative scheduling techniques described in Chapter 6 do not suffer that much loss in IPC.

2.4.4 Indexed Schemes

Several papers describe schedulers which attempt to remove the associative search needed for tag broadcasts by using reverse pointers, effectively replacing the CAM with a RAM [13, 14, 48, 49, 65, 68]. With reverse pointers, instructions keep track of their dependants rather than their source operands. Weiss and Smith [68] described the first indexed scheduler called Direct Tag Store (DTS). The scheduler in this paper uses a table with one entry for each instruction that points to a dependent instruction. Only that one instruction can be directly awakened. If an instruction has more than one dependant, the issue of the latter dependant is stalled. Follow-on techniques have described implementations where more than one dependant may be immediately awakened [14, 39, 48, 49, 65]. These papers also introduced new techniques for handling the situation where an instruction has more dependants than can be held in the RAM. Canal and González [13, 14] also compared performance when stalling issue to placing excess dependants in either an in-order or out-of-order small window. Sato and Arita [65] also compared performance of stalling issue to placing excess dependants into an in-order queue. Önder and Gupta [48, 49] handled this situation by adding additional dependences between two “sibling” instructions with the same source operand. Huang et al. [39] investigated stalling issue as well as reverting back to broadcasting to excess dependants.

The biggest problem with indexed schedulers is that additional hardware support is required for branch misprediction recovery. It is difficult to recover before the branch is the oldest instruction in the machine. Rather than just flushing particular entries of the window, as with traditional schedulers, the entries of instructions that are not flushed must be modified to remove information about off-path dependants. Additionally, this style of wakeup logic does not perform well when instructions have more than one dependant, which is frequently the case. The techniques for handling additional dependants require either adding new scheduling structures, increasing the latency of the RAM, or increasing the program’s critical path.

2.4.5 New Circuit Designs

The paper by Henry et al. [34] discusses how to simplify the associative searches required in conventional wakeup logic by attaching cyclic segmented prefix (CSP) circuits to the reorder buffer. The scheduling latency is reduced when using a logarithmic implementation of a CSP circuit for wakeup. The problem with this design is that it requires all reorder buffer entries to be a part of the scheduling window logic. Simulation results in Chapter 4 show that the scheduling window can have half as many entries as the instruction window without an impact on IPC. Furthermore, the amount of wakeup logic scales linearly with the number of architectural registers, which may be undesirable for some ISAs.

The paper by Goshima et al. [32] describes a scheduling implementation that uses dependence matrices. It is similar to the array scheduler used in Chapter 6, and will be described in Section 6.2.3.

Chapter 3

Simulation Infrastructure

3.1 Baseline Microarchitecture Model

The baseline microarchitecture used in this dissertation is a superscalar out-of-order processor designed to exploit instruction-level parallelism. Figure 3.1 shows the stages of the pipeline. Each stage may consist of one or more cycles, although each stage is fully pipelined. The thick lines represent the boundaries of the in-order and out-of-order portions of the pipeline. Instructions are fetched and decoded in the first 5 cycles. The rename stage identifies instructions' register dependences and translates architectural register identifiers into physical register identifiers. Because this stage includes dependence analysis and steering for 16-instruction fetch packets, it takes 10 cycles. After instructions are renamed, they are then *issued* to the scheduling window. The scheduler is responsible for determining when an instruction is ready to execute and scheduling it on a functional unit. After being scheduled for execution, instructions read the Payload RAM and physical register file to get any source operands that are not read from the bypass network. After all instructions in the oldest fetch packet have completed execution, they can retire. After they retire, their re-

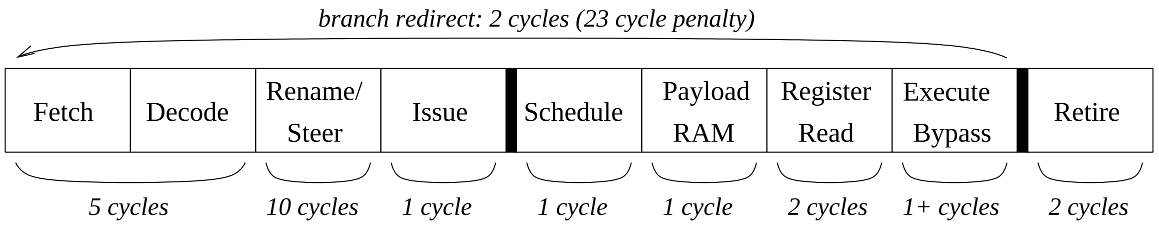


Figure 3.1: Processor Pipeline

sources (except for the youngest instance of each architectural register) are freed and may be used by subsequent instructions. A block diagram of a 4-cluster configuration of the processor datapath is shown in Figure 3.2.

3.1.1 The Pipeline Front-End

The fetch mechanism can fetch up to three basic blocks (where a basic block is defined as a group of instructions ending in a control flow) per cycle. Some experiments have used a trace cache, and that will be indicated when those experiments are discussed. The instruction cache is a 4-way set associative 64KB cache with 64-byte lines. It takes five cycles to fetch and decode the instructions before they reach the rename stage.

The conditional branch predictor is a hybrid conditional branch predictor [43] indexed by a combination of branch addresses and branch history. The predictor contains two pattern history tables, gshare [43] and PAs [70], each 16KB in size and containing 64K 2-bit saturating counters. The selection mechanism is a table of 64K 2-bit saturating counters indexed by a hash of the branch address and global history. In addition to the conditional branch predictor, there is a 4K-entry, 4-way set associative BTB, a 32-entry call-return stack, and a 64K-entry indirect branch predictor.

3.1.2 Register Renaming and Dependence Analysis

In the Rename stage, each instance of an architectural register is assigned an entry in the physical register file. Dependence analysis logic identifies the physical register identifiers of an instruction's source operands. It also determines where those source operands will be produced or where they reside. They may either already reside in the register file at the time the dependent instruction is renamed and issued, or they may be broadcast at a later time. Additionally, the renaming logic may determine if the instructions producing its source operands have already been scheduled for execution, or if the dependent instruction

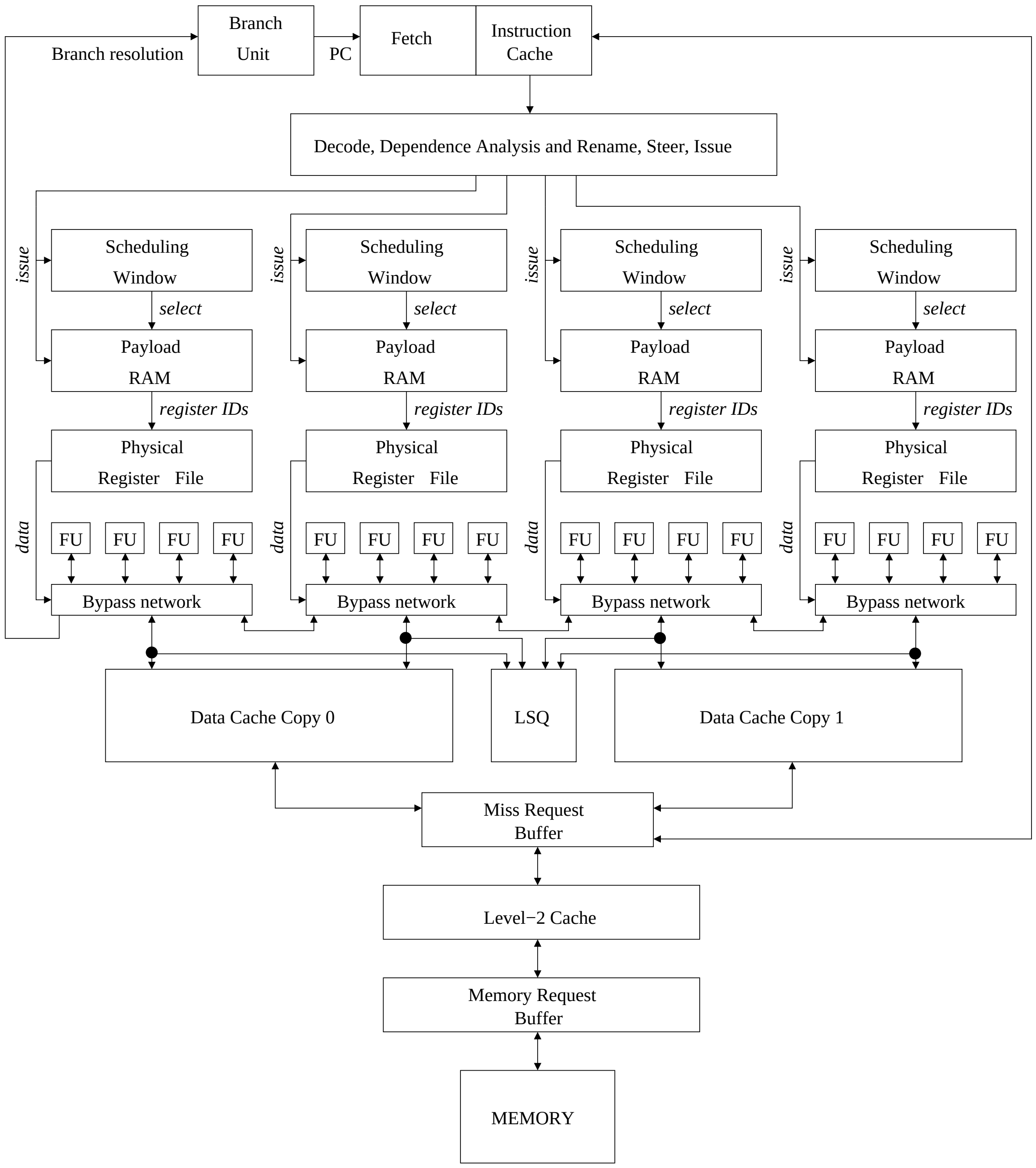


Figure 3.2: Processor Datapath

must wait for the tags of its source operands to be broadcast.

The Register Alias Table (RAT) is a table with one entry for each architectural register that contains a mapping from the architectural register to an entry in the physical register file. During renaming, an instruction reads the RAT for each architectural source register to obtain the physical register identifier for that source. It also writes the identifier of its allocated physical register into the rename map entry associated with its architectural destination register.

In a superscalar processor, all instructions in a fetch packet are renamed in the same cycle. To detect dependencies between instructions in the same packet, the sources of each instruction are compared to the destinations of all previous instructions in the packet. If an instruction's parent (i.e. the instruction producing its source operand) is in its packet, the identifier of the physical register allocated to the parent overrides the identifier obtained from the rename map. Figure 3.3 shows the dependency analysis logic for the first three instructions in a packet. On the left side are the physical register identifiers that were outputs of the RAT. These will only be used if there are no subsequent instructions writing to the same architectural registers. The architectural register identifiers of each instruction's source and destination operands are listed across the top. For each source operand, there is one comparator for the destination operand of every older instruction. The priority circuits select the youngest instruction with a matching register identifier.

Due to the fact that it is a wide-issue pipeline, the instruction renaming phase takes 10 cycles. Renaming logic can be pipelined by assigning physical destination registers in advance using a free pool of physical registers [33, 55]. The dependence analysis logic may be pipelined as well. For example, the logic on the upper-right side of the dotted line is the dependence analysis logic which sets the mux controls. This logic may be pipelined over multiple cycles without increasing the number of comparators as long as the RAT is still accessible in one cycle. The comparator logic is pipelined over five cycles in this model.

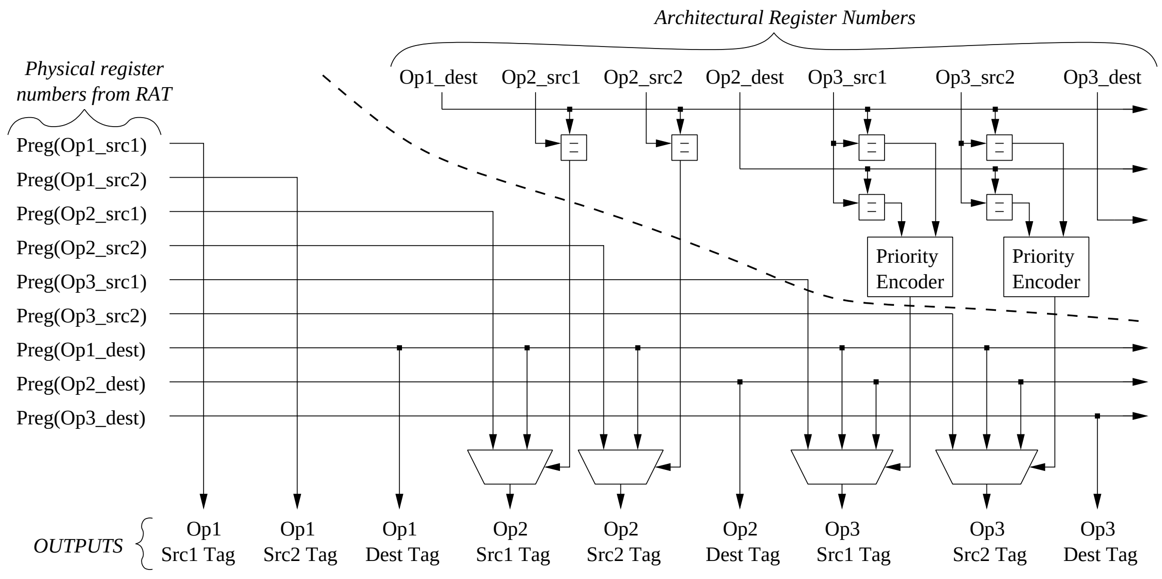


Figure 3.3: Dependency Analysis Logic for Three Instructions.

In the latter stages of the Rename phase, entries in the instruction and scheduling windows are allocated, and instructions are steered to particular locations within the execution core.

3.1.3 Instruction Window

After instructions are renamed, they are issued into the instruction window. The instruction window holds up to 512 instructions. Each entry contains information an instruction needs to determine when it can retire as well as what action to take if a misprediction or exception occurs. Instructions reside in the instruction window until they are retired. All instructions in a fetch packet are retired atomically. If the fetch packet contains a mispredicted branch or an excepting instruction, then the wrong-path instructions and excepting instructions are squashed. It is possible to squash part of a fetch packet. Instructions in a fetch packet can retire (or be squashed, in the case wrong-path instructions or excepting instructions) when all previous fetch packets have retired and all instructions have completed execution. The issue stage is assumed to take only 1 cycle because much of the work required to allocate instructions in the instruction and scheduling windows can be overlapped with the rename stage.

3.1.4 Instruction Scheduling

After instructions are renamed, they are placed into the scheduling window at the same time that they are placed in the instruction window. Unlike the instruction window, instructions reside in the scheduling window only until they have finished execution. The scheduling window for the baseline machine holds up to 256 instructions[1]. Each entry only holds information an instruction needs to be scheduled for execution (no data).

The baseline wakeup logic is implemented using CAMs. Each entry of the scheduling window keeps track of the tags of an instruction's source operands and whether or not those operands are available. In conventional processors, the tags are the same as the physical register identifiers, although this does not have to be the case. Figure 3.4 shows information associated with one entry of the scheduling window. The SRC TAG fields contain the physical register numbers of the source operands. The DELAY field indicates the latency (or predicted latency) of the source operands. The M (Match) bits indicate if the source tags have been broadcast. The COUNT fields are used as countdown timers. When a source tag is broadcast, the M bit is set and the COUNT field begins counting down the

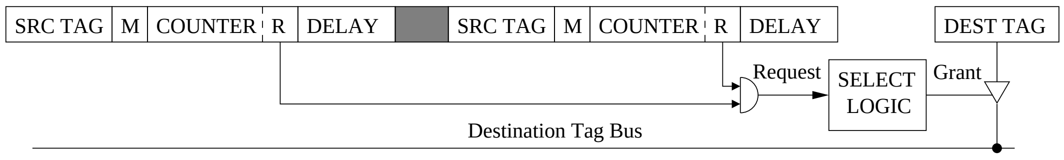


Figure 3.4: A Scheduling Window Entry

[1]Simulations in Chapter 4 show that the scheduling window can be much smaller than the instruction window without a large impact on IPC, since many instructions do not stay in the scheduling window very long.

cycles of the parent instruction's latency, as indicated by the DELAY field. After the latency of the source operand instruction has been counted, the R (Ready) bit is set. [2] Once both Ready bits are set, the instruction may be selected for execution. The select logic consists of a priority circuit. The output of the priority circuit is used to gate the destination tag (stored in a RAM) onto one of the Destination Tag Buses.

The scheduling window is built as a random access structure. In other words, entries may be allocated and de-allocated out-of-order. It is not implemented as a priority queue because collapsible buffers consume a large amount of power [12]. Typically one or more instructions are selected for execution each cycle, so with collapsible buffers, the instructions in the queue younger than the selected instruction must be shifted down. This results in high dynamic power consumption from the switching activity. Implementations of oldest-first selection priority exist for this type of scheduler implementation [12]. However, simulations of our baseline machine show that using select logic in which priority is hardcoded by an instruction's location in the window lowers IPC by less than 0.7% compared to perfect oldest-first priority. Oldest-first priority will be used in all experiments unless otherwise indicated.

After being scheduled for execution, instructions read the Payload RAM. The Payload RAM is a table with one entry for each instruction in the scheduling window. It contains information that an instruction needs to execute such as its opcode, immediate

[2] Note that this is only one of many possible implementations. For example, if each functional unit only executes instructions that all have the same latency (or predicted latency), the tag broadcast can simply be delayed so that it occurs a fixed number of cycles before the result broadcast. This eliminates the need for the DELAY, M, and COUNTER fields. However, if functional units can execute instructions of differing latencies, this solution is unsuitable at high clock frequencies: Multiple pipeline stages may need to broadcast tags, rather than just one. Either these pipeline stages would need to arbitrate for tag buses, or the number of tag buses would need to be increased.

data values, and the source operand physical register numbers[3]. No decoder is needed for this RAM because the select logic selects the rows to be read.

3.1.4.1 Busy-Bit Table

Before an instruction is issued into the window and allocated a scheduling entry, it must know how to initialize the M, READY, and COUNT fields. If an instruction is issued after one of its source operand tags was broadcast, the corresponding M bit should be set, and the COUNT field and READY bit should be initialized to the correct values according to how may cycles ago the tag was broadcast. If the instruction must wait for the tag to be broadcast, the READY and MATCH bits should be initialized to 0 and the COUNTER field should remain the initial value determined by the producer instruction's latency.

The Busy-Bit Table [69], shown in Figure 3.5, is used to retrieve information about source operand availability. This table must be accessed after an instruction's source operands have been renamed but before it is placed into the scheduling window. The BBT has one entry for each physical register to indicate if that register value has been produced. In actual implementation, it may contain MATCH, COUNT, and READY fields just like the scheduling logic.[4] The countdown timer begins counting down when the instruction writing to that physical register has been scheduled to execute on a functional unit. When an instruction is issued, it reads the BBT entries corresponding to the physical registers of its source operands, as well as the tag buses. If a source operand's BBT entry is set or

[3]If the source operand physical register numbers were read out of the CAMs, then the Payload RAM could be accessed in parallel with the register fi le. However, additional ports would be needed to read these values. That would increase the latency and power of the register fi le. In deep pipelines, he performance decrease of adding a pipeline stage to access the Payload RAM is less than the performance decrease of increasing the latency of the scheduling window. Data is shown in Chapter 4.

[4]As with the scheduling logic, alternative and simplifi ed implementations exist. It is possible to build this table with a single bit per entry (hence the name *Busy Bit Table*) but for purposes of explanation we will discuss only the general case that supports instructions of multiple latencies.

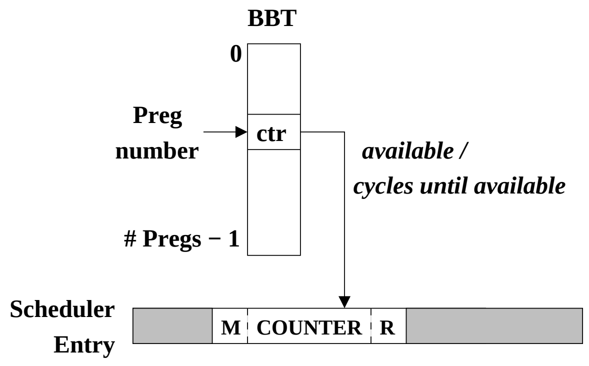


Figure 3.5: Busy-Bit Table Initializing Scheduler Entry

its tag is broadcast in that cycle, then the instruction sets the corresponding COUNTER or READY bit in its scheduler entry. A BBT entry is initialized when a new instruction is assigned to the corresponding physical register.

Rather than using a BBT, it is possible to add the information stored in this table to the RAT. However, this alternative implementation is more complicated. Since the RAT is indexed by architectural register rather than physical register, the RAT would have to be augmented to include a CAM in which the physical register identifier for each entry is compared against all tag broadcasts. When physical register tags are broadcast, a match would occur if the corresponding architectural register had not subsequently been renamed. Additionally, as with the conventional RAT, support for branch misprediction recovery is required. For these reasons, a BBT is used in the baseline model.

3.1.5 Physical Register File

The physical register file is a separate structure from the instruction and scheduling windows. In a processor with maximum execution width N, a conventional unified physical register file would have N write ports and $2N$ read ports, assuming each instruction can have a maximum of one destination register and two source registers. This number of ports is needed only for the worst case, and rarely are all ports used in the same cycle. However, reducing the number of write ports would require the scheduling logic to support arbitration for the ports. However, the baseline processor does use techniques to reduce the number of write word lines and read ports. These techniques are described below.

The register file used as a baseline for this dissertation reduces the number of write word lines (thereby reducing the area, latency, and power) by using Register Write Specialization, first published by Seznec, Toullec, and Rochecouste [61]. The register file is divided into sets in order to reduce the number of write word lines per bit cell. For example, suppose there are 512 registers, divided into four sets. The first set contains physical register numbers 0 to 127; the second set contains physical register numbers 128 to 255; and so on. Assuming a traditional register file has N write ports, each entry of each set of a write-specialized register file would have only $N/4$ write ports. However, an additional restriction is placed on instruction execution. An instruction's set is identified by its physical destination register number. Only $N/4$ instructions in each set may write to the register file (and hence execute, unless buffers are used) in the same cycle. While this requires support by the scheduling logic in an unclustered machine, this constraint is inherent when the functional units are clustered because each cluster has its own scheduling logic. Even on an unclustered machine this technique has little impact on IPC, however. Simulation of the unclustered baseline model used in this dissertation showed that incorporating register write specialization lowers IPC by less than 0.14% on average and 1% in the worst case for the SPEC2000 integer benchmarks. It reduces the area of the physical register file by 75%, assuming the area is a function of the number of word lines and bit lines running over each bitcell. The size of the write word line decoders is reduced as well, since two fewer bits are needed as input.

A second technique is also used to reduce the area (and hence latency and power) of the register file. The register file is replicated 4 times in order to reduce the number of read ports. Each replica has 8 read ports and is used by instructions executing on a group of 4 functional units. Even though there are now four times as many bits of state total, the access latency and power are reduced since each replica occupies considerably less area.

As a result of these two techniques, each register bitcell has 12 word lines (eight for reading, four for writing) and 24 bit lines (eight for reading, 16 for writing), as opposed to 48 of each, as a centralized register file would have. For the unclustered model, this replication technique and register file write specialization combined reduce the register file latency by 65%.

3.1.6 Execution

The baseline microarchitecture contains 16 functional units which can execute any instruction, although there are further limitations on the number of loads and stores that can execute per cycle. Instruction replay occurs if more loads are selected for execution than there are free cache ports. Execution latencies are shown in Table 3.1. Most multi-cycle operations are pipelined. Loads take a minimum of three cycles (assuming a hit in the L1 cache or load-store buffer).

Instruction Class	Latency (in Cycles)
integer arithmetic	1
integer multiply	8, pipelined
fp arithmetic	4, pipelined
fp divide	16
loads and stores	3 minimum
store to load forwarding	3 minimum
all others	1

Table 3.1: Instruction Latencies

3.1.7 Memory System

The data cache is a 4-way set associative 64KB cache with 64-byte line size. The cache is divided into 16 banks, each with one write and two read ports. Although previous work has examined the use of a partitioned cache with a clustered execution core [57], the baseline architecture here simply assumes there are two identical copies of the data cache to reduce access latency. The data cache latency is two cycles, and load and store instructions take an additional cycle for address computation. Perfect memory dependence prediction is assumed since previous work [18, 46, 47, 66] has shown that memory dependence predictors can be used to obtain performance comparable to that of a machine with perfect memory dependence prediction.

Misses in both the Level-1 instruction and data caches are placed in the memory request buffer. This buffer contains 32 entries, and memory accesses generated from several instructions may all be piggybacked on the same entry. Different types of memory requests are prioritized in the following order, from highest to lowest: instruction fetches, data fetches, data stores, instruction and data prefetches, and lastly, write-backs. A request accesses the Level-2 cache when a port for its desired bank is available. The L2 cache is an 8-way set associative 1MB cache with 64-byte line size. It is divided into eight banks, each with one read and one write port. The access latency is a minimum of 10 cycles. When a memory request hits in the L2 cache, the instructions associated with that miss are notified.

Memory requests that miss in the L2 cache are placed in a miss request queue. Instruction fetches and write-backs have the highest priority in this queue, followed by data loads and stores, followed by prefetches. Requests may access the desired memory bank when it is available. The main memory is modeled as having 32 banks interleaved on 64-byte boundaries, with a minimum access latency of 100 cycles.

3.2 Evaluation Methodology

Results are shown for the SPEC2000 integer benchmarks, as these benchmarks are sensitive to the microarchitecture of the execution core. Whenever possible, an input set provided by SPEC was used. However, in many benchmarks, some of the input files were truncated in order to reduce simulation time. The truncated inputs were designed to have execution behavior similar to the behavior when using the original inputs. A summary of the benchmarks and the input sets used are shown in Table 3.2. Because no profiling was performed, only one input set per benchmark was used for all experiments.

Benchmark Name	Description	Input set
gzip	compression algorithm	ref.graphic (trunc)
vpr	FPGA circuit placement and routing	place.in (test)
gcc	C compiler	jump.i
mcf	combinatorial optimization	inp.in (test)
crafty	chess game	crafty.in (trunc)
parser	English text processing	test.in (trunc)
eon	computer visualization	test inputs
perlbmk	PERL interpreter	perfect.pl (test)
gap	group theory, interpreter	test inputs (trunc)
vortex	object-oriented database	test inputs (trunc)
bzip2	compression algorithm	input.source (trunc)
twolf	Place and Route Simulator	test inputs

Table 3.2: SPEC CPU2000 Integer Benchmarks

3.3 Performance and Power Models

3.3.1 Performance Simulator

The performance results in this dissertation were produced using a simulator (scarab) for the Alpha ISA developed by members of the HPS research group [17]. It is a cycleaccurate, executable-driven simulator that models wrong-path events. The simulator does not model system calls; they are executed on the host machine.

3.3.2 Power Simulator

3.3.2.1 Overview of Wattch

The power model is based on the Wattch toolset [7]. Wattch has a set of functions that compute capacitances of on-chip structures given a microarchitecture definition, defined by the user, and the physical circuit characteristics, defined by the Cacti [62] toolset. Wattch uses these capacitance estimates along with switching activity computed by the performance simulator to model dynamic power consumption.

The physical design parameters are calculated based on the feature size (transistor length). The user can define the feature size in a header file of the simulator. Given the feature size, physical design parameters are computed using information contained in the Cacti [62] header files. Examples of physical design parameters are the capacitance for a given wire length for different metal layers, the minimum wire pitch for each layer of metal, and supply voltage.

The user must specify the microarchitecture definition. Wattch provides a sample microarchitecture definition that is similar to the one used by the *sim-outorder* model of SimpleScalar [11], although Wattch also provides functions with which the user can define a new microarchitecture. These functions calculate the capacitance of common building blocks such as RAM and CAM arrays, register files, decoders, latches, wires (per unit length), and the select logic used by the instruction scheduler. The user must specify the number of entries and ports for each structure, along with other specific information needed for certain types of structures. Clock power was estimated from the Alpha 21264 [2], but is scaled according to physical parameters, clock frequency, and area estimates. Combinational logic is difficult to model with Wattch. Power estimates for functional units, control logic, and decode logic were based on previous publications [2, 5, 71].

Switching activity is monitored by interfacing Wattch to the performance simulator. Event counters are added to the performance simulator (scarab in this case) to obtain this information. These counters are then used by the Wattch model to measure dynamic power consumption.

3.3.2.2 Modifications to Wattch

The Wattch model has been heavily modified to work with our performance simulator and accurately represent each processor model studied in this dissertation. This section describes some of the modifications.

First, a new microarchitecture definition, the one previously described in this chapter, is used. The functions for estimating the maximum power consumption of the basic processor building blocks (arrays, CAMS, control logic and other combinational logic, wires, and clock distribution) are taken from Wattch, although they are scaled appropriately to match the performance model. Power models for additional microarchitecture features not present in the original Wattch model (e.g., the memory request buffers, multi-ported instruction cache and L2 cache, branch prediction structures, two copies of the data cache, latches for additional pipeline stages, and all other structures mentioned in this chapter) have been added. Many of these structures can be implemented as a simple RAM and/or CAM, some with and some without decoders. The capacitance of the register file was recomputed to model a replicated register file using Register Write Specialization. This was accomplished by recomputing the bit line and word line lengths given the configuration described in Section 3.1.5.

Some assumptions about floorplanning were made for the machine configurations which model a clustered core. These floorplanning assumptions are used to model capacitances of wires running between clusters as well as capacitances of tag and data buses within clusters. Groups of four functional units (assuming four functional units per cluster) are stacked vertically so that the data bit lines are interleaved. The register file sits directly above the functional units, muxes, and the latches which hold data being read from and written to the register file, in addition to the data that was broadcast from other clusters. The width of each cluster is a function of both functional unit area estimates [50] as well as the maximum number of bit lines at any point in the datapath for wide-issue clusters. We conservatively assume that this width is constrained by the width of the register file. The result bus from each functional unit runs vertically within its own cluster to the register file write latch, as well as horizontally to the other clusters and then vertically across all other stacks of functional units as well. In the clustered models, the tag and data buses that are received from other clusters are also considered when calculating the number of word lines and bit lines for each structure in a cluster. Section 4.4 will discuss alternative designs which have differing numbers of word lines and bit lines for the register files and scheduling windows.

Most of the event counters and the functions for measuring dynamic power consumption (as a fraction of maximum power consumption) have been changed from those present in the original Wattch framework. The modified power model distinguishes between different types of accesses to many structures. For example, data cache reads and writes do not consume equal amounts of power in the model. The most obvious difference is due to the fact that the cache is duplicated in order to reduce the access latency by halving the number of read ports to each copy. A write, from either a store instruction or a cache-line fill, must update both copies of the cache.

Register file writes also have a disproportionate power dissipation compared to reads [72]. The primary distinction is that the register file bit cells in this model have two bit lines for each write port and one bit line for each read port [24]. As a result, register file reads and writes are modeled as different types of accesses.

The conditional clock gating style called *CC3* in Wattch is modified as follows: unused ports dissipate at least 19% of their maximum power even when they are not accessed, and data bit lines that do not switch still dissipate power even when the port is active. This overhead is a very rough estimate of leakage power, although it is not a very precise estimate because leakage power is primarily determined by transistor features rather than wire capacitance.

Chapter 4

Clustering

Clustering the execution core reduces the time to execute a chain of dependent instructions as long as all instructions in the chain execute in the same cluster. Clustering reduces the execute-bypass loop latency by reducing the forwarding latency between functional units within the same cluster. As the number of functional units per cluster is reduced, the forwarding latency in the cluster is reduced and the clock frequency can be increased. However, the IPC may go down because of load balancing problems and extra cycles required to forward values between clusters. The biggest frequency gains can be made by using clusters of size one (one functional unit per cluster), but this configuration also suffers from the most inter-cluster communication delays. By bitwise interleaving two or more functional units that perform simple ALU operations, it may be possible to build clusters with more than one functional unit with little or no increase in the wire delays incurred when forwarding the output of one functional unit to the input of the other. The cluster configurations studied in this dissertation use two to four functional units per cluster.

Clustering may also be used to reduce power consumption. However, depending on the amount of loss in IPC and changes in critical loop latency, there may or may not be a reduction in power consumption. This chapter explains several factors that may affect performance and power of a clustered processor. Section 4.1 examines the reasons that clustering lowers IPC – specifically, the causes of resource contention and inter-cluster communication delays. Section 4.2 examines the scheduling window and register file latencies of several clustered configurations. Section 4.3 examines how clustering affects power consumption. The clustering paradigm in these first three sections uses a register file that is replicated across all clusters. Any register data that is produced is broadcast to all clusters.

Section 4.4 examines two alternative clustering paradigms in which register values are not broadcast to all clusters when they are produced. The first, called PART, uses a partitioned register file like the one described by Canal, Parcerisa, and González [16, 52]. The second, called Selective Cluster Receipt (SCR), uses a replicated register file, although register values are only sent to the register files and functional units of the clusters that contain dependent instructions. The PART model reduces the size of the inter-cluster bypass network, but the SCR model does not. Section 4.4 examines the window latency of the PART model and compares the power and performance of similarly sized PART, SCR, and baseline clustered models.

4.1 Effects of Clustering on IPC

This section analyzes the effects that cause IPC degradations in clustered processors relative to an unclustered processor. These effects include inter-cluster communication delays and load balancing. Inter-cluster communications may increase the delays between producer and consumer instructions, which will increase the length of the program's critical path. As illustrated by Figures 1.2 and 1.3, this is becoming a bigger problem as execution width increases. With poor load balancing, instructions in a given cluster may be delayed due to lack of available functional units in that cluster.

Seven machine models are investigated in this section. They are identified in Table 4.1. The processor model parameters common to all models was discussed in detail in Chapter 3, but the relevant model parameters and the differences between these models will be described below.

The processor pipeline in all models can fetch, issue, and execute up to 16 instruc-

Name	Steering	Scheduling Window	Functional Units
CS-CF	front-end	partitioned	clustered
FlexCS-CF	front-end	partitioned, 4 or 8X as big as CS-CF	clustered
CS-16CF	front-end	partitioned	16 per cluster (max 16 exec width)
US-CF	schedule time	unified	clustered
US-UF	n/a	unified	unified
US-UF+1	n/a	unified	unified 1-cycle delay between ALL units
US-UF+2	n/a	unified	unified 2-cycle delay between ALL units

Table 4.1: Summary of seven models.

tions per cycle. The instruction window consists of all instructions that have been issued (placed into one of the scheduling windows) but have not yet retired. Up to 512 instructions can be in the instruction window. Instructions are selected for execution from the scheduling window, which holds only those issued instructions that have not successfully completed execution. The instruction window contains all information necessary to retire instructions in program order, while the scheduling window contains only information necessary to schedule instructions for execution. The size of the scheduling window, not the instruction window, is what determines the scheduling latency. The baseline model, called **US-UF** (for Unified Scheduling window, Unified Functional units) uses a single 256-entry scheduling window with no clustering of functional units.

In the first clustered model, called **CS-CF** (for Clustered Scheduling window, Clustered Functional units), the scheduling window and functional units are divided equally among all clusters. This model will also be referred to as the baseline clustered model throughout this dissertation. Two configurations of this model are evaluated: the first with four clusters each containing four functional units and a 64-entry scheduling window, and the second with eight clusters each containing two functional units and a 32-entry scheduling window. There is a 2-cycle inter-cluster forwarding delay in the 4-cluster configuration. In the 8-cluster configuration, there is a 1-cycle delay between two clusters within an adjacent pair of clusters (i.e. clusters 0 and 1; 2 and 3; 4 and 5; 6 and 7), and 2-cycle delay between functional units in different pairs of clusters.

The remaining models in this section are not intended to be considered for actual implementation; they are used just for this clustering behavior study. In the second clustered model, called **FlexCS-CF**, the total scheduling window size is still the same (256 entries) as that of the unclustered model, **US-UF**, although there is no limit on the number of instructions in any one scheduling window. In other words, it is possible that one scheduling window contains 256 instructions, but the others would all have to be empty. It allows us to see the effects of issue stalls or undesirable cluster assignments that appear in the baseline clustered model when a particular cluster's window is full. For example, if memory latency or too few execution units are a performance bottleneck (as may be the case for easily predictable instruction streams) then a particular scheduling window may be full, although other windows may have free space. The **FlexCS-CF** model does not have this restriction. It is desirable to steer an instruction to the cluster containing its producer. If the desired scheduling window is full, the instruction may be steered to an alternate cluster and incur an inter-cluster forwarding delay penalty. With this model, no "alternate" steering assignments needs to be made. An instruction can always be steered to its desired cluster unless the instruction window is full.

The third model, called **CS-16CF** also uses clustered scheduling windows and functional units. The difference between this model and **CS-CF** is that each cluster contains 16 functional units. The total execution width of the machine is still limited to 16 instructions, however. With this model, up to 16 instructions may execute per cycle, but those instructions may be distributed among the clusters in any permutation. For example, 16 instructions all residing within one cluster may execute in the same cycle, but no instructions from other clusters could execute in that cycle. When compared to the **CS-CF** model, this model allows us to see the effects of functional unit contention on IPC.

The fourth model, called **US-CF** uses a unified scheduling window, but the functional units are partitioned into clusters. Instructions are dynamically assigned to a cluster at schedule time from the unified window as follows: when an instruction's last-arriving source operand broadcasts its tag, the instruction is scheduled immediately on the same cluster as its last-arriving source operand if a functional unit in that cluster is available. If no functional unit in that cluster is available, it must wait until the following cycle to be scheduled. The instruction may execute in other clusters when the result is available in those clusters. If there is no contention for functional units in the source operand's cluster, it will be scheduled on that cluster. When there is contention, the scheduler gives priority to the oldest instructions.

The **US-CF** model is used to analyze the impact that steering at schedule-time rather than issue-time can have on IPC. When steering at issue time, instruction execution latency (or at least the latency of memory operations) is not known. This information is known later after source operands have been scheduled. This model should not be used as an upper bound on clustering or steering algorithm performance for several reasons. First, there are no limitations on load balancing. For example, it is theoretically possible that all instructions could execute on only one cluster (although the maximum IPC would be limited to the cluster width), which may not be possible with clustered paradigms that have a partitioned scheduling window. Second, it does not have oracle information about the future critical path of the program, and hence cannot make optimal cluster assignments.

The last two models are called **US-UF+1** and **US-UF+2**. These are the same as the model with the unified execution core but with an extra 1- or 2-cycle data forwarding delay for all execution results. With the **US-UF+1** model, the scheduling logic may be pipelined over two cycles since all instructions take a minimum of two cycles to execute. With the **US-UF+2** model, the scheduling logic may be pipelined over three cycles. Alternatively, these models could represent a worst-case steering scenario in which all instructions are delayed by inter-cluster forwarding penalties.

In all of the models with clustered scheduling windows described above, instructions are *steered*, or assigned to a cluster, during the renaming process. High-performance steering algorithms were used with all models, but a detailed discussion of these algorithms is deferred until Chapter 5. In brief, all models used *dependence-based* steering algorithms which try to send instructions to the same clusters as their source operands in order to reduce the number of inter-cluster bypasses.

Figure 4.1 shows the IPC for six models, all except **US-UF+1**, with the 4-cluster configuration[1], and Figure 4.2 shows the results of all seven models with the 8-cluster configuration. For the 4-cluster configuration, the fully clustered model (**CS-CF**, the set of bars second to the right) shows a 23% IPC improvement over the model with a minimum 2-cycle execution and scheduling latency (**US-UF+2**). For the 8-cluster configuration, the **CS-CF** model has an IPC 17% higher than **US-UF+2**, and 1% higher than **US-UF+1**. These comparisons demonstrate that many of the worst-case clustering delays were avoided by steering instructions to the same cluster as their source operands. It also demonstrates that if an instruction window is so large that the scheduling logic must be pipelined over two or more cycles, it is better to divide the window into clusters.

The model that increases the size of each scheduling window (**FlexCS-CF**) performs only about 1% better than the **CS-CF** model. This indicates that alternate steering

[1]Results for the **US-UF+1** model are not shown for the 4-cluster confi guration because the 4-cluster confi guration has a two-cycle delay between clusters.

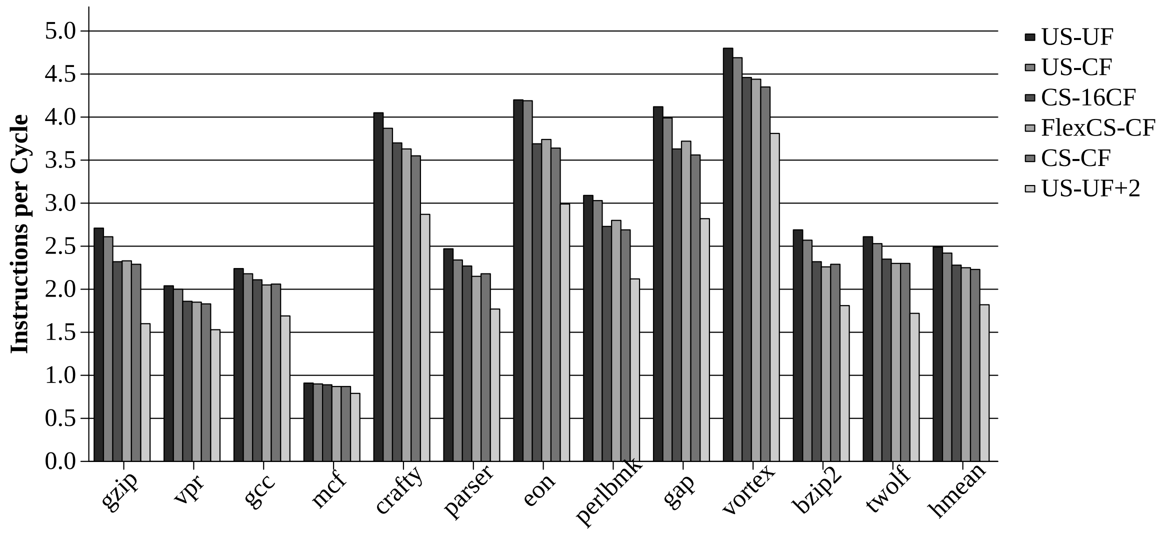


Figure 4.1: IPC for baseline and 4-cluster configurations.

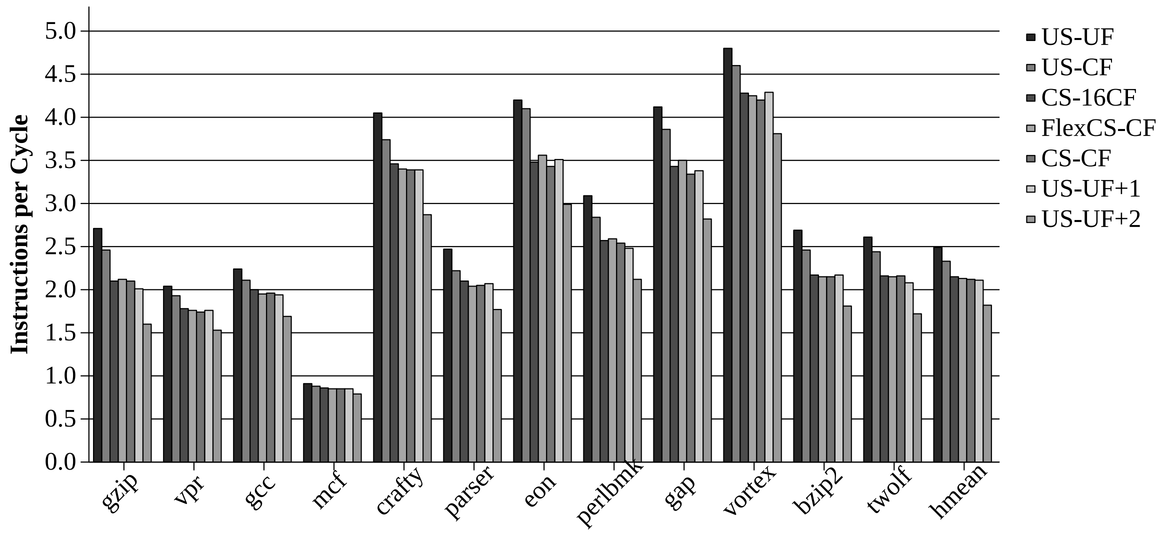


Figure 4.2: IPC for baseline and 8-cluster configurations.

assignments due to full scheduling windows have little overall effect on IPC, provided that the steering algorithm assigns instructions to an alternate cluster (as opposed to stalling issue) when the desired cluster is full. The effect of full windows will be discussed further in Chapter 5, when steering algorithms are evaluated.

The model that allows up to 16 instructions in a given cluster to be scheduled (**CS-16CF**) performs about 2% better than the **CS-CF** model, indicating functional unit contention when clustering has a small effect on IPC. However, inter-cluster bypass delays have a much bigger impact on IPC. This is also true for the 8-cluster configurations.

The improvement in IPC of the unified model (**US-UF**) over the fully clustered model (**CS-CF**) is 11.6% and 17.5% for the 4-cluster and 8-cluster configurations, respectively. Of the experimental clustered models, the biggest gains in IPC are made when cluster assignments are made at schedule time, as with the **US-CF** model.

Figure 4.3 shows a breakdown of where the last source operand comes from for all instructions. The left and right bars for each benchmark show the breakdown for the fully-clustered 4- and 8-cluster configurations, respectively. The top portion of each bar represents the fraction of instructions that either have no source operands or do not need to wait for any source operands to become ready before they can be scheduled for execution. On average, this is 22% of all instructions. For the 4-cluster configuration, half of all instructions, on average, received their last source operand from their own cluster, and the remaining 28% of all instructions had to wait to receive their last operand from another cluster. When using 8 clusters, 34% of instructions received their last source operand from the same cluster, while 44% received their last source operand from another cluster. While the fraction of instructions waiting for their last source operand to be forwarded from another cluster is highly dependent on the steering algorithm, the data presented here is representative for a high-performance dependence-based steering algorithm. Chapter 5 will use this metric, along with others, to understand the behavior of steering algorithms.

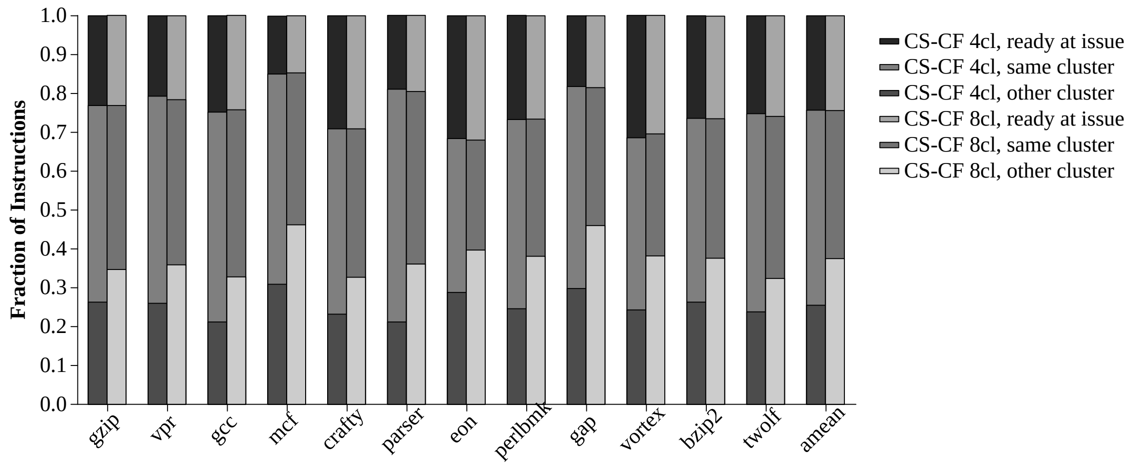


Figure 4.3: Location of last arriving source operand for all instructions.

4.2 Effects of Clustering on Latency

This section discusses how clustering and partitioning can affect the scheduling and register file access latencies. These effects are measured for the baseline clustered model, which replicates the register file on all clusters. The scheduling window is evenly partitioned among all clusters. Sections 4.2.1 and 4.2.2 discuss how latency changes as parameters are scaled. Section 4.2.3 provides data (Cacti estimates) for the scheduling window and register file configurations discussed in this section.

4.2.1 The Scheduling Window

The scheduling window was modeled as a CAM structure with two tags per instruction entry. The latency of the scheduling loop is the sum of the select logic latency and the scheduling window latency. Scheduling window latency depends on the following parameters:

1. **Machine execution width.** When execution width increases, more instructions must be selected for execution each cycle and more instructions broadcast their tags each cycle. Each tag bus is a read port for the CAM. Increasing the number of ports means increasing the number of bit lines and word lines needed for each CAM bitcell. This means that increasing the number of tag buses in the scheduling window increases the area of the scheduling window as well as the amount of comparator logic needed for CAM matches. This increases both the latency and power of the scheduling window.

2. **Tag buses per cluster.** For an unclustered processor, the number of tag buses is the same as the execution width. For a clustered processor, the number may still be equal to the total machine execution width if there are point-to-point connections between all scheduling windows and functional units for tags and data. This is the case for the baseline clustered model. However, in some clustered designs, particularly the models with partitioned register files described in Sections 2.1.1 through 2.1.3, only a subset of register values may be sent between clusters. In these models, the number of tag and data buses may be reduced, although this is not necessary just for implementation. Reducing the number of buses will reduce the scheduling window latency, area, and power, although it requires arbitration for the buses. Arbitration may increase inter-cluster forwarding latency.

3. **Number of entries per scheduling window.** The number of instructions that can reside in each window affects the amount of ILP that may be exploited. Also note that clustered configurations that require copy instructions to send data between clusters may need additional entries per window to hold the copy instructions.

4. **Number of issue (i.e. write) ports per window.** The number of instructions that can be allocated to a given cluster in a given cycle is determined by the number of issue ports. The scheduling latency is longer with a larger number of issue ports

because the area is increased. However, this also means that average communication delays may be reduced because the steering algorithm has more opportunity to steer an instruction to its desired cluster. The performance impact of changing the number of issue ports is discussed in detail in the Steering chapter in Section 5.1.1.

4.2.2 The Register File

1. **Machine execution width.** The number of register file ports is determined by the execution width. For the baseline, we assume two read ports and one write port for each instruction that can begin execution in the same cycle. Clustering may be used to reduce the number of ports by using register files local to each cluster.

2. **Cluster execution width.** In the baseline clustered model, the execution width of a cluster affects the number of read ports but not the number of write ports, as all results are broadcast to all clusters by default. In clustered designs introduced later in this chapter, the number of write ports may be limited to the execution width of the cluster plus a few more write ports needed to receive data forwarded from other clusters.

3. **Number of entries.** Models with partitioned register files need fewer entries in each local register file. However, as with the number of entries in the scheduling window, these models may need additional register file entries total since some register values may be duplicated across clusters. For example, with 4 clusters, each cluster may need more than N/4 entries (where N is the number of entries in an unclustered register file).

4. **Number of write ports.** The number of write ports to each register file is the same as the number of scheduling window tag buses because every register tag broadcast

is associated with a data broadcast that occurs a few cycles later in conventional out-of-order processors.

5. **Effect of register write specialization.** By using the optimization described in Sections 2.1.4 and 3.1.5, the number of write word lines is reduced by a factor of 4. This reduces the area of the register file by 75%, and the latency by 45%.

4.2.3 Latency Estimates

Latency estimates were obtained using Cacti [60] with 0.09μm feature size. For the scheduling window, Cacti is used to model the tag broadcast and tag match (i.e. wakeup) latency, but not the latency of the select logic. The window is modeled as a CAM structure with two CAM tags per instruction. Cacti was slightly modified to model register files with Register Write Specialization as follows: the decoder and bit line delays were computed by using one fourth the number of write word lines as in a conventional register file. There was no modification to the word line delay because of the assumed layout of the register file: all four banks are stacked vertically such that all bit lines run across all four banks.

The latency estimates are intended to make the fairest possible comparisons between the various clustered configurations examined later in this chapter. Tables 4.2 and 4.3 show latencies obtained from Cacti of the scheduling window (tag broadcast and wakeup latencies combined) and register file while scaling the parameters discussed above. The columns of the table, from left to right, are the total execution width, the number of clusters, the total number of scheduling window entries for all clusters combined, the number of write (i.e. issue) ports for each scheduling window, the scheduling window (wakeup and tag broadcast) latency, and the register file latency. The number of entries in the register file is held constant at 512. Each cluster contains a copy of all 512 register file entries, although the scheduling window is partitioned among the clusters. The main observations from these tables are that increasing the number of ports to either the scheduling window or the register file has a much larger impact on latency than increasing the number of entries. When dividing the execution core into four clusters, the scheduling window latency can be reduced by as much as 55%. The register file latency is reduced by 40%. Partitioning the execution core into 8 clusters can provide further reductions in latency.

Exe Width	Num Clusters	S.W. Entries	S. W. W Ports	Window Latency	Reg File Latency
16	1	192	16	1.66	1.08
16	1	256	16	2.02	1.08
16	1	384	16	2.84	1.08
16	1	512	16	3.74	1.08
16	4	192	4	0.90	0.64
16	4	192	6	0.93	0.64
16	4	192	8	0.96	0.64
16	4	192	12	1.02	0.64
16	4	192	16	1.08	0.64
16	4	256	4	1.01	0.64
16	4	256	6	1.05	0.64
16	4	256	8	1.08	0.64
16	4	256	12	1.16	0.64
16	4	256	16	1.23	0.64
16	4	384	4	1.25	0.64
16	4	384	6	1.30	0.64
16	4	384	8	1.36	0.64
16	4	384	12	1.47	0.64
16	4	384	16	1.59	0.64
16	8	256	4	0.78	0.58
16	8	256	6	0.80	0.58
16	8	256	8	0.82	0.58
16	8	256	12	0.86	0.58
16	8	256	16	0.90	0.58

Table 4.2: Scheduling Window and Register File Latency estimates, 16-wide execution.

Exe Width	Num Clusters	S.W. Entries	S.W. W Ports	Window Latency	Reg File Latency
16	8	384	4	0.90	0.58
16	8	384	6	0.93	0.58
16	8	384	8	0.96	0.58
16	8	384	12	1.02	0.58
16	8	384	16	1.08	0.58
4	1	192	4	1.00	0.39
4	1	256	4	1.13	0.39
4	1	384	4	1.41	0.39
8	1	192	8	1.21	0.57
8	1	256	8	1.41	0.57
8	1	384	8	1.85	0.57
8	2	192	4	1.05	0.46
8	2	192	6	1.10	0.46
8	2	192	8	1.16	0.46
8	2	256	4	1.21	0.46
8	2	256	6	1.28	0.46
8	2	256	8	1.35	0.46
8	2	384	4	1.56	0.46
8	2	384	6	1.15	0.46
8	2	384	8	1.21	0.46
8	4	192	2	0.77	0.41
8	4	192	4	0.80	0.41
8	4	192	6	0.83	0.41
8	4	256	2	0.84	0.41
8	4	256	4	0.88	0.41
8	4	256	6	0.92	0.41
8	4	384	2	1.00	0.41
8	4	384	4	1.05	0.41
8	4	384	6	1.10	0.41

Table 4.3: Scheduling Window and Register File Latency estimates, contd.

4.3 Effects of Clustering on Power Consumption

Even though the register file is replicated in our baseline clustered architecture, power consumption when clustering, as compared to an unclustered processor, is still reduced because the number of read ports to the register file and the number of issue ports for each scheduling window are reduced. This is because register file area typically grows with the square of the number of ports (assuming a wire-bound structure). Hence, by reducing the number of ports to each copy, total power consumption is reduced.

Figure 4.4 shows the breakdown of the relative power consumption for three 16-wide machines (shown left to right in the figure): the first with a unified execution core, the second with four clusters, and the third with eight clusters. Figure 4.5 shows the IPC of these models. Switching from a unified core to four clusters results in a 12.7% reduction in IPC on average, but a 19% reduction in power[2]. Switching from four to eight clusters results in an additional 4.2% reduction in IPC and 4.3% reduction in power. The clustered configurations shown are not the highest-performing nor the lowest-power configurations. They were selected because they showed the best performance for a given power consumption[3]. Additional configurations are shown later in this section. Note that the same clock frequency was assumed in all designs, although the clustered designs may run with a higher frequency core, increasing both performance and power consumption. The power consumption breakdown is explained in detail below.

The power consumption results are shown relative to the average power consumption of the unified machine. The breakdown of the power consumption for each microar-

[2]These experiments use no modifi cations in clock frequency. The scheduling window, register fi les, and execution units of the clustered core can either be designed with slower transistors for additional power savings, or a higher clock frequency for performance improvement

[3]The four-cluster confi guration has six issue ports per scheduling window. The eight-cluster confi guration has three ports per scheduling window. More ports would improve the performance at the cost of higher power consumption.

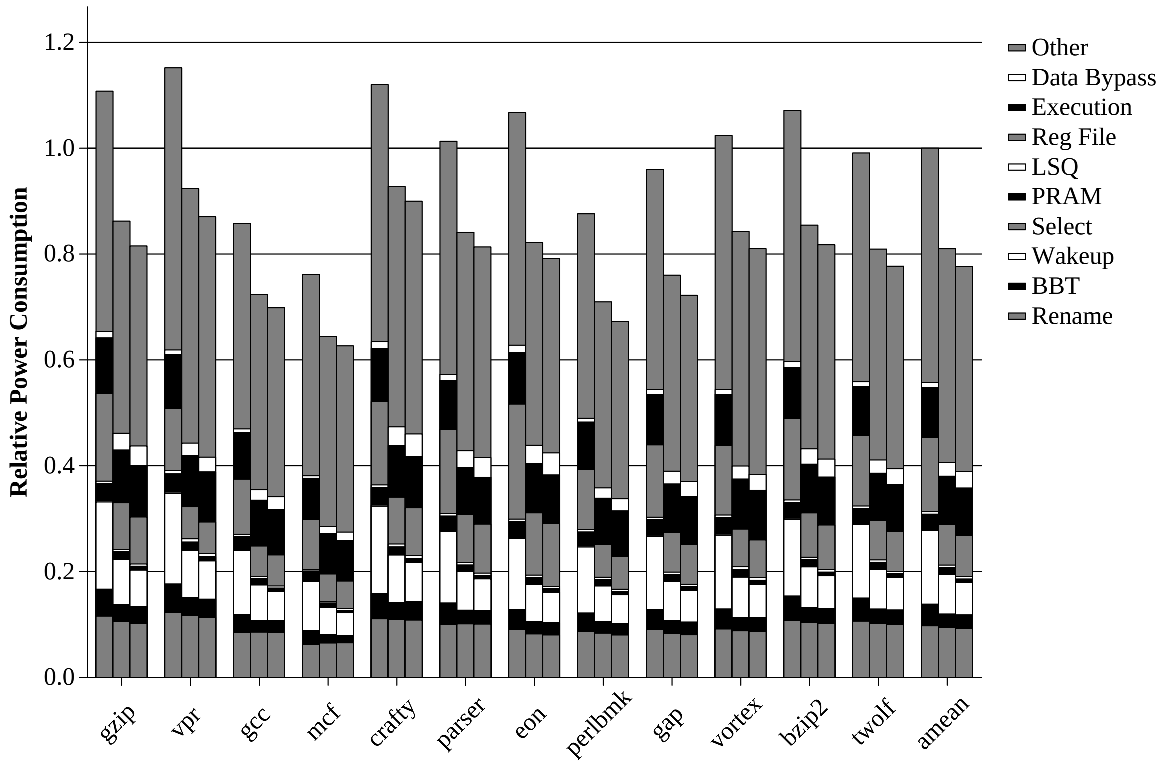


Figure 4.4: Relative power consumption for 1, 4, and 8-clustered models (shown left to right).

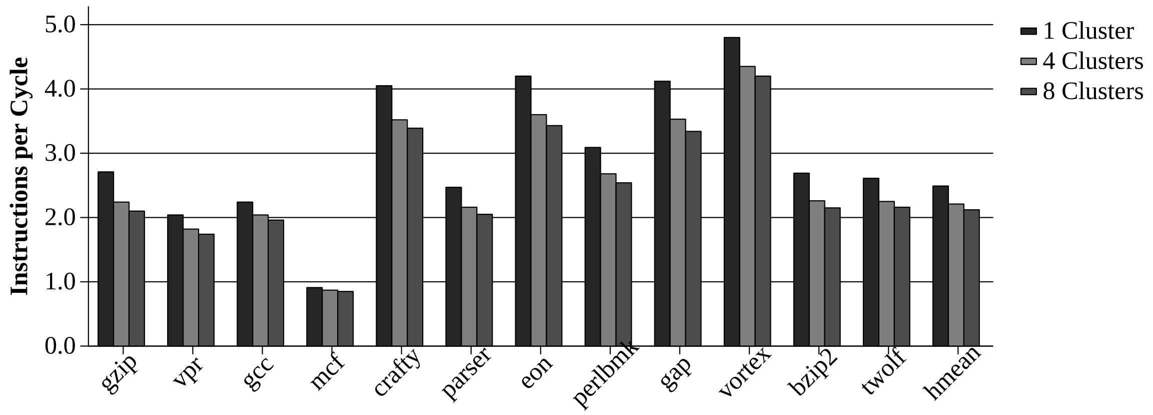


Figure 4.5: IPC for 1, 4, and 8-clustered models.

chitectural unit is shown within each bar. The power of the **Rename** unit consists of the Register Alias Table along with all of the dependency analysis logic. The power of the RAT makes up about 40% of the Rename power in these models; the rest is consumed by the comparators and muxes. The next unit, the **BBT**, is replicated among each cluster in the clustered configurations. This reduces the power consumption since each replica has fewer read ports. The power of the **Wakeup** logic is shown in white above the BBT power. The wakeup power is significantly reduced in the clustered machines because the scheduling window is partitioned, and each cluster has fewer than 16 issue ports. The **Select** logic is a very small component of total power consumption and is not visible in the figure. The **Payload RAM** is a table (one per cluster) that holds all information that instructions need to execute other than source operand data. There is one entry per instruction in the scheduling window, and each entry contains 20 bits. When clustering, the Payload RAM has lower power consumption because each partition has fewer ports. Each cluster's partition has as many write ports as the scheduling window has issue ports, and as many read ports as the number of instructions that can be selected for execution from within the cluster. The 64-entry **Load-Store Queue** consumes a relatively small amount of the total power consumption. The bars for its power consumption, shown in white, are barely visible. The **Register File** consumes considerably less power in the clustered models even though it is replicated across all clusters. This is because the number of read ports to each replica is significantly reduced. There is no change in the number of write ports, however. The **Execution** power is fairly constant among all configurations. The **Data Bypass** network, shown in white, consumes more power in the clustered configurations because of the inter-cluster forwarding buses and the considerably longer wires required to route results to any cluster. In the baseline clustered model, there is no reduction in the number of data bypasses, so all functional unit results are forwarded to all replicas of the register file. The **Other** category consists of branch prediction structures, memory request buffers, the instruction, data, and Level-2 caches, and clock tree power. Because the power consumption of these units does not change much among the different configurations (aside from minor variances due to different levels of performance), they are combined into one bucket in the figures to make them easier to read. Power consumption is not constant among benchmarks, but it does correlate with the amount of both on-path and off-path instructions executed per cycle.

The remaining results in this section will only show results for the 4-cluster configuration, as the 8-cluster configuration shows a more significant IPC loss.

Figure 4.6 shows the effect of scaling the number of issue ports for each scheduling window on the total power consumption. All models have 4 clusters, and the number of issue ports per cluster for each configuration, shown from left to right, are 4, 6, 8, 12, and 16. The improvement in IPC, shown in Figure 4.7, from using 16 ports does not make up for the increased power consumption. High-performance steering algorithms were used with all configurations. Steering algorithms, when considering the number of ports per cluster, are discussed in detail in Chapter 5.

Figures 4.8 and 4.9 show the relative power consumption and IPC when scaling the number of scheduling window entries. All of these configurations have 4 clusters with 8 issue ports per cluster. Using more than 256 entries does not seem to show much improvement in IPC. The power consumption continues to increase, however. Based on these results, the baseline clustered model has scheduling windows with a total of 256 entries, or 64 entries per cluster.

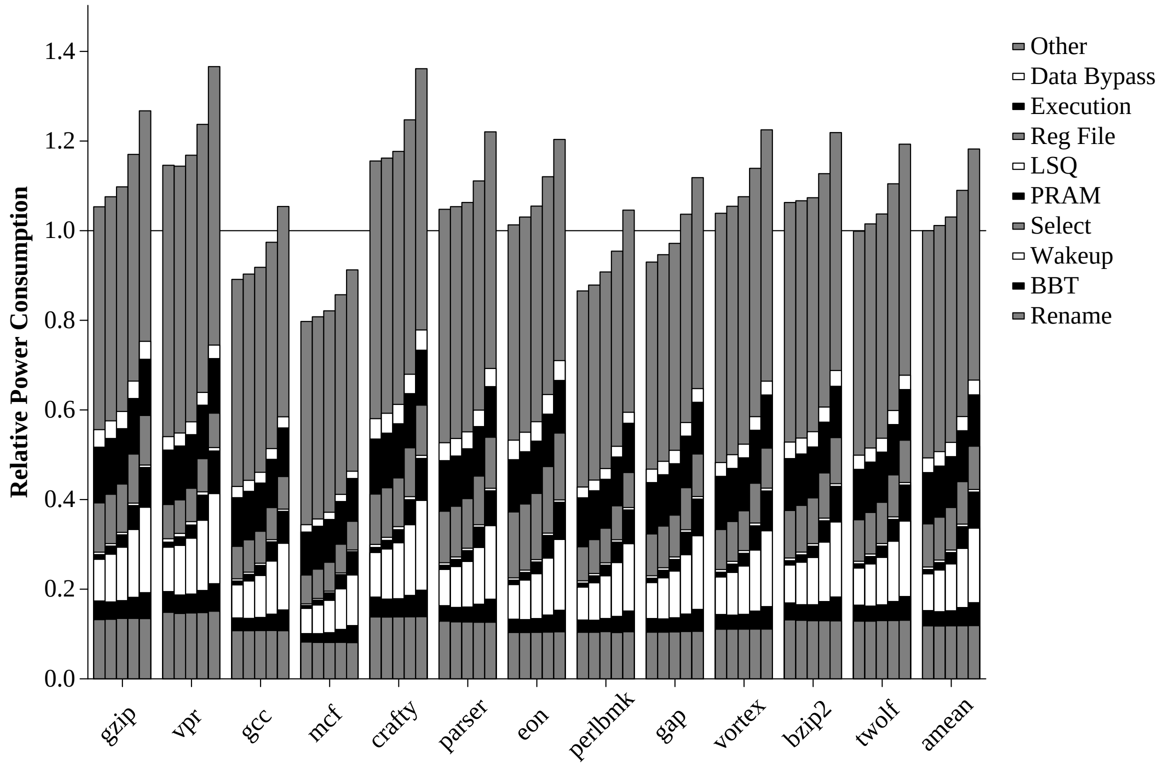


Figure 4.6: BASE model power consumption while scaling the number of issue ports per cluster: 4, 6, 8, 12, 16 ports.

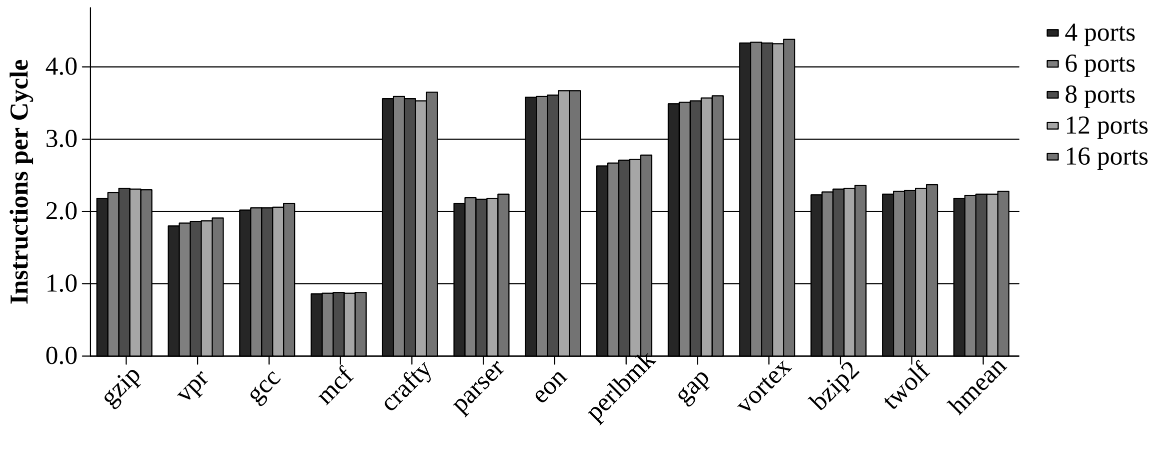


Figure 4.7: BASE model IPC while scaling the number of issue ports.

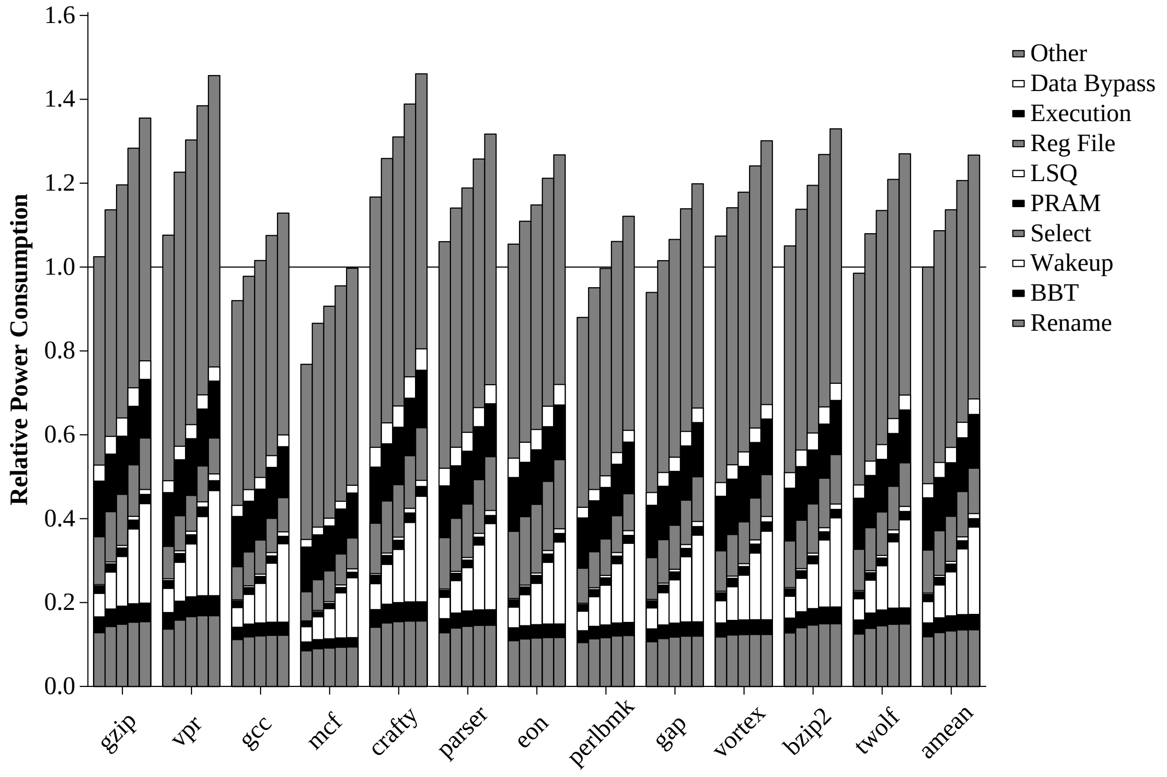


Figure 4.8: BASE model relative power consumption when scaling the number of scheduling window entries (128, 192, 256, 384, 512).

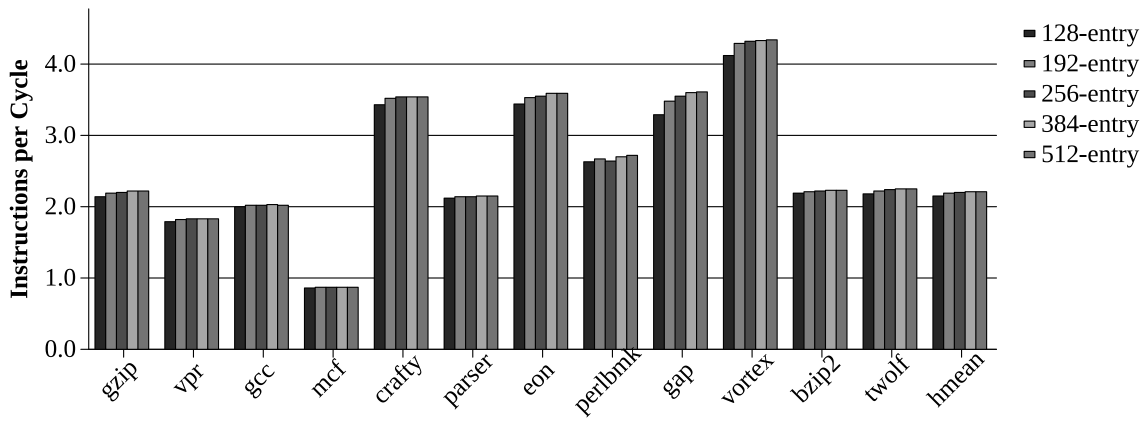


Figure 4.9: IPC when scaling the number of scheduling window entries.

4.3.1 Summary

Section 4.2 showed that clustering is an effective technique to reduce latency of structures in the execution core. By replicating the instruction window and register file across clusters, the number of ports can be reduced. This reduces the latency for each replica. Section 4.3 showed that replication can even reduce power consumption. This is because area grows with the square of the number of ports. Hence, replicating a structure can reduce area if enough ports in each copy are eliminated.

The performance results demonstrated that no more than 256 scheduling window entries are needed given an instruction window size of 512. Furthermore, for a designer targeting a specific cycle time, it may be better to increase the number of ports at the cost of reducing the number of entries.

4.4 Replication vs. Partitioning

The previous performance and power results of clustered designs shown in this chapter have used register files that were replicated across all clusters. Each cluster contained copies of all 512 entries of the physical register file. This section compares clustered paradigms with a replicated register file to those with a partitioned register file.

With a partitioned register file, a given register value may reside in only one cluster, or it may reside in multiple clusters if it is needed by instructions in multiple clusters. Register values do not have to be forwarded to clusters where they are not needed. This means that each register file partition may have a reduced number of entries and write ports. Because all register values are not stored in all clusters, the scheduling tags associated with the instructions producing those values do not have to be broadcast to all clusters. This means that each scheduling window may have fewer tag buses and fewer entries. The latency savings that result from using partitioned register files are shown in Section 4.2.

When limiting the number of tag buses and write ports in each cluster, a mechanism for arbitration must exist to handle contention for available buses and ports. This mechanism could use additional broadcast buffers [73], or transfer or copy instructions [15, 23, 36]. When using copy instructions, values may be sent between clusters using either global buses or a point-to-point network. As observed by Parcerisa et al. [53], global buses shared by multiple clusters slow down the inter-cluster forwarding in a machine with several clusters because the arbitration for the buses must be centralized. Centralized arbitration can be avoided with a point-to-point network. That is, each cluster has a dedicated path to each other cluster. Copy instructions must be scheduled within their own cluster to be transfered on an inter-cluster bypass path. Register values forwarded from other clusters may be stored in either transfer buffers [23, 36] or within the local register file partition.

A model using a partitioned register file is described in more detail in Section 4.4.1. Section 4.4.2 describes an improved clustered design with replicated register files using a technique called *Selective Cluster Receipt.* Power and performance of both of these models is discussed in Section 4.4.3.

4.4.1 Partitioned Register File Model

The PART model assumes that values are moved between clusters via copy instructions. A point-to-point bypass network is used to avoid the centralized arbitration unit. The bypass network assumes full connectivity between functional units within the same cluster, although there is limited connectivity between functional units in different clusters in order to reduce the number of register file write ports and scheduling tag buses. Each cluster has at least one bypass path to each other cluster, and clusters do not share bypass paths[4]. This

[4]It is possible for clusters to share bypass paths. For example, one cluster can use a bus in even cycles and one can use it in odd cycles. However, this severely limits IPC because the average inter-cluster forwarding delays are increased.

means each scheduling window has enough tag buses for all instructions issued within its own cluster, plus one or more additional buses for each other cluster. Assuming the minimum connectivity (each cluster has one path to each other cluster), for a 16-wide machine with four clusters, this means there are seven tag buses per window, rather than 16, as used in the baseline clustered architecture.

Register values forwarded from other clusters are stored in the register file rather than in a separate buffer. This eliminates the need for a separate transfer buffer that instructions must read for the location of their source operands, although it places additional pressure on the register file.

The register alias table (RAT) and renaming logic must be augmented to support multiple mappings for each architectural register. With four clusters, there may be up to four mappings of each architectural register. When an instruction renames its source operands, the RAT indicates which clusters contain a valid mapping of their source operands. If the instruction's cluster does not contain a valid mapping, then a *copy instruction* must be inserted into the scheduling window containing a valid copy of the source operand, and an additional physical register must be allocated in the consuming instruction's cluster. Copy insertion requires additional write ports to the scheduling window and additional physical register file entries for best performance. The power model models multiple copies of the RAT (with fewer read ports per copy) because this technique reduces power consumption.

To determine the optimal design points, the partitioned model is evaluated while scaling the number of register file entries, window entries, the number of bypass paths between clusters, the number of issue ports to each scheduling window, and the total number of RAT ports. The experimental results of IPC and power consumption of selected design points are shown in Section 4.4.3.

4.4.2 Replication with Selective Cluster Receipt

Selective Cluster Receipt (SCR) (formerly called *Demand-Only Broadcast* [9, 10]) is a technique which uses a replicated register file with enough ports to support maximum throughput but does not always broadcast register results within all clusters, thereby saving dynamic power consumption. In a conventional clustered processor with a replicated register file, register values are received by all clusters. With selective cluster receipt, an instruction's result is only received by the remote clusters if it is needed by dependants within those clusters. For the configurations presented in this dissertation, this eliminates 66% of the register file writes and bypasses within clusters. Unlike the PART model, tags are always broadcast to all clusters.

This section discusses the implementation of Selective Cluster Receipt. Experimental results of its performance and power savings are shown in Section 4.4.3.

4.4.2.1 Justification and Grounds for SCR

The primary aim of SCR is to have a clustering paradigm that avoids the penalty associated with inter-cluster bypass arbitration while getting the lower power consumption that results from fewer bypasses and register file writes. Arbitration increases the length of the execute-bypass loop between clusters. Previous works use one of two techniques for handling this arbitration. Both are explained below.

The first arbitration technique was described by Hrishikesh [36]. With this arbitration technique, each producer has information indicating where its dependants reside. This information is stored in a table in each cluster called the *Inter-cluster Forwarding Bit Table* (IFB Table). This table indicates, for each physical register, to which clusters that register value should be forwarded. This table is read after instructions are selected for execution so that they know where to broadcast their tags. The first problem with this approach is that the table lookup increases the inter-cluster scheduling latency (the time to forward tags between clusters), and possibly intra-cluster scheduling latency as well (depending on the specific implementation). This is the increase in execute-bypass latency referred to earlier. The IFB Table must be updated by the steering logic after instructions are assigned to clusters. The steering logic would have to inform each cluster (probably every cycle) of any dependants that were issued to the other clusters. This information is not available in a timely manner because of a delay between the steering and scheduling logic, called the *detect-to-set delay* by Hrishikesh [36]. If a producer instruction were scheduled for execution after its dependant was steered to another cluster, then the table lookup would indicate that the producer had a consumer in another cluster – the one containing the consumer. For instance, if a dependent instruction is issued just a few cycles before, at the same time as, or after the producer instruction is scheduled, the information arrives too late. The producer instruction would not have broadcast its tag to the other clusters because the IFB Table had not been updated by that point in time. Hrishikesh [36] investigates several techniques to avoid this problem, but IPC is lowered as compared to the second arbitration technique, discussed below.

The second arbitration technique, and the one used by the PART model, is to use copy instructions rather than IFB Tables. Copy instructions are issued to the producer's cluster regardless of whether the producer instruction has been scheduled, so there is no possibility of deadlock as long as the copy instruction actually gets inserted before its consumer[5]. Copy instructions add a cycle of arbitration latency to the execute-bypass loop between clusters. This is because they add a node in the dependence chain between producers and consumers in different clusters, and they take a cycle to be scheduled after the producer has been scheduled.

[5]If an instruction is issued before a copy instruction on which it is dependent, the copy may get stalled because of a full window. If all instructions in the window were dependent on stalled copies, deadlock would occur. Hence copies must be issued before their dependants.

Since the aim of SCR is to avoid increasing the length of critical loops, both of these approaches are undesirable. An alternative implementation, and the one used by SCR, is to forward the scheduling tag to all clusters, as in the baseline model. Hence there is no arbitration. Each cluster contains a table indicating which register values produced by instructions in other clusters are needed by instructions in that cluster. In fact, a table indexed by physical register number already exists in each cluster – the BBT. This table can be used to determine if a given cluster holds consumers of a particular register value. Each cluster decides whether or not to accept a register value forwarded from another cluster based on the contents of its own BBT. Hence the name *Selective Cluster Receipt*. This technique avoids the timing problems that would have existed if the inter-cluster forwarding was blocked from within the producing cluster using an IFB Table.

4.4.2.2 Implementation of SCR

When using Selective Cluster Receipt, an instruction's result is not received by the register file and functional units within a cluster unless that cluster holds a consumer at the time that the instruction's tag is broadcast. Each cluster contains a copy of the Busy-Bit Table (BBT), just as the baseline clustered model does. However, rather than just one bit for each BBT entry, there are two: the *Broadcast bit* indicates if the tag has been broadcast to that cluster, just like the single bit in the original BBT entry. The *Use bit* indicates if there are any instructions within that cluster requiring that physical register. BBT entries are reset when an architectural register is first mapped to a physical register, just as in the conventional design.

Table 4.4 shows the possible states of a BBT entry along with their meanings. A summary of all possible state transitions of a BBT entry is given in Figure 4.10. This diagram is shown in the form of a Mealy machine. The arcs indicate both the input action that causes a BBT entry to be accessed, as well as the output action that results from reading

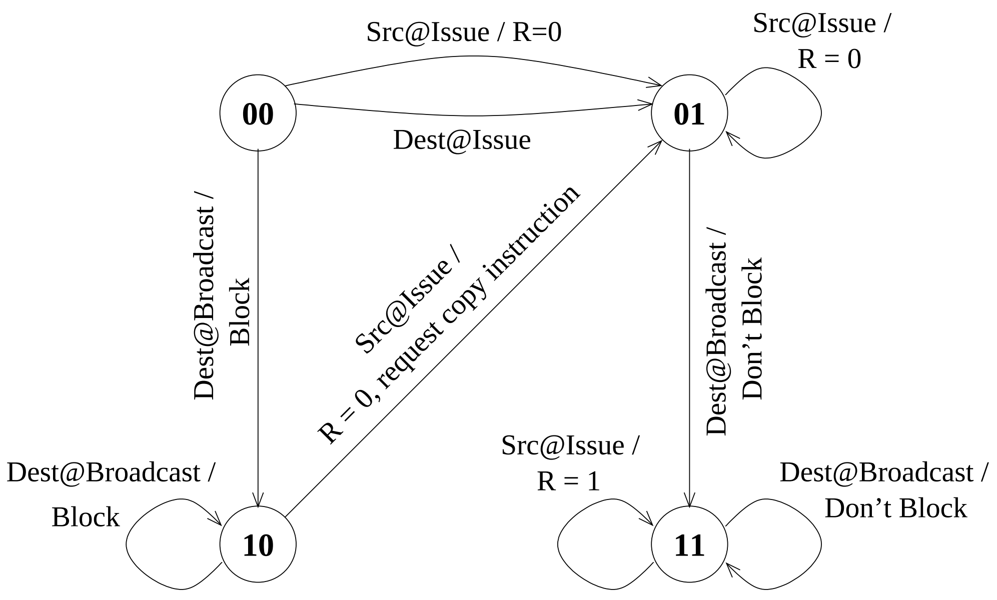
B.C.	USE	Status
0	0	Tag not yet broadcast. Data will not be received.
0	1	Tag not yet broadcast. Data will be received.
1	0	Tag was broadcast, but data was / will not be received.
1	1	Tag and data were both broadcast; result is available.

Table 4.4: Possible states of a BBT entry.

the table. The input actions are the times which the BBT is accessed: at issue time, when the entries for source and destination operands are accessed, and at tag broadcast time, when the entry for the destination operand is accessed. The output actions are either initializing the Ready bits, which is done at issue time, or controlling a cluster's receipt of a register value, which is done at tag broadcast time. These state transitions are explained below.

When an instruction is first placed into a cluster, it reads the BBT entries corresponding to its source operands. If both the Use and Broadcast bits of a source operand's entry are set, then the source operand is available and the corresponding Ready bit in the instruction's reservation station entry is set. If the Broadcast bit is clear but the Use bit is set, then either the instruction producing that source operand is waiting to be scheduled in that cluster, or there is an older instruction in that cluster waiting for the result to be broadcast, but it has not been broadcast yet. After reading the contents, it sets the Use bits of the entries corresponding to its destination and source registers.

If the Broadcast bit is set but the Use bit is clear when the BBT entry of a source operand is read, that means the instruction producing that value has already broadcast its tag, but the data value was not received in that cluster because there were no previous instructions in that cluster requiring the value. A copy instruction must be inserted. The mechanism for inserting copy instructions is discussed in detail in Section 4.4.2.3. Additionally, the Broadcast bit of that entry must be cleared, and the Use bit is set.



Key:

State bits: {XY} = {Broadcast bit, Use bit}
"Src", "Dest": reading BBT entry for source or destination operand
"@Issue", "@Broadcast": at Issue or Broadcast time
"R": Ready bit of wakeup logic
"Block": refers to blocking data receipt by a cluster

Figure 4.10: Mealy diagram for a BBT entry.

When an instruction’s tag is broadcast to a cluster, the Broadcast bit for its destination register is set, just as in the baseline. Additionally, it reads out the value of the Use bit. If the Use bit is set, the instruction’s result will be broadcast to the register file and functional units in this cluster. If the Use bit is not set, the data broadcast will be blocked by that cluster. The logic for blocking the broadcast may add one gate to the data path, depending on the implementation. Note that the controls for blocking a data broadcast are located entirely within a cluster, so that no arbitration or acknowledgment signal between clusters is required to block the broadcast.

There is plenty of time to set the controls to block the data broadcast and prevent the register file write. This is because normally the instruction's data would be broadcast N cycles after its tag is broadcast (assuming N is the number of pipeline stages between the last scheduling stage and the last execution stage for a majority of integer instructions). When the Use bit is read, it will enable the latch for the data result bus N cycles later. N is generally at least as large as the minimum number of cycles for the register file access plus execution, and will increase as pipeline depths increase. As an example, N is 5 cycles in the Intel Pentium 4 [35].

Table 4.5 gives an example in which instruction *A* in Cluster 0 produces a value needed by instruction *B*, which is issued to Cluster 1. BBT-0[A] refers to the BBT entry in Cluster 0 corresponding to A's destination register, and BBT-1[A] refers to the BBT entry for *A* in Cluster 1. In this example, an instruction's result is broadcast 3 cycles after its tag, and there is a 2-cycle delay between the two clusters. Initially, *A* resides in Cluster 0 but *B* has not yet been issued to Cluster 1's window. The Use bit for *A* in BBT-0 is set because instructions always write back to their own cluster. In Cycle 0, *A* broadcasts its tag to Cluster 0. In cycle 1, instruction *B* is issued to cluster 1. *B* sets the USE bit of A's BBT entry in that cluster to indicate that there is an instruction in Cluster 1 that will need A's result. Since *B* is issued to Cluster 1 before *A*'s tag is broadcast to that cluster, no copy instruction is needed. When *A*'s tag is broadcast within Cluster 1 in Cycle 2, BBT-1 is read to see if there are consumers of *B* in that cluster. Since there are, *A*'s data result broadcast is not blocked.

4.4.2.3 Copy Instructions

In the previous example, if instruction *B* were issued to cluster 1 after *A*'s tag was broadcast within that cluster and the Use bit for *A*'s BBT entry in Cluster 1 (BBT-1[A].Use) was not set, then *A*'s data would not be broadcast to that cluster in cycle 5. In this situation, a *copy instruction* is required to re-broadcast the result. The copy instruction is inserted

Cycle	Action		BBT-0 BC	BBT-0 USE	BBT-1 BC	BBT-1 USE
Init	*A* is in Cluster 0.	A		**X**		
		B				
0	*A* is selected, broadcasts tag	A	**X**	**X**		
	to cluster 0.	B				
1	*B* issued to cluster 1.	A	**X**	**X**		**X**
	B reads Use bit of source operand.	B				**X**
	B sets Use bit for its sources and destination.					
2	*A*'s tag is broadcast to cluster 1.	A	**X**	**X**	**X**	**X**
	Read BBT-1[A].Use. It's set, so *A*'s tag will	B				**X**
	be broadcast 3 cycles later. *B* wakes up.					
3	*A*'s data broadcast to Cluster 0.	A	**X**	**X**	**X**	**X**
	B broadcasts its tag in Cluster 1.	B			**X**	**X**
5	*A*'s data broadcast to Cluster 1.	A	**X**	**X**	**X**	**X**
	B broadcasts its tag in Cluster 0.	B	**X**		**X**	**X**

Table 4.5: Example of inter-cluster forwarding with Selective Cluster Receipt.

into the cluster that produced the source operand (although it could actually be inserted into any cluster that didn't block the broadcast). After being scheduled, it reads the register file and re-broadcasts the physical register destination tag and the data, similar to a MOVE instruction with the same physical source and destination register.

In order to detect if a copy instruction is needed, when an instruction is first issued and reads the BBT entry of its source operand, it must read out the old contents before it is set, like a scoreboard. If the Use bit is clear and the Broadcast bit is set, then a copy instruction must be inserted, and the instruction's Ready bit is not set.

In each cluster there is a bit-vector specifying which physical registers require copy instructions to re-broadcast the data. All instructions issued to a cluster may set bits of this bit-vector. If an instruction reads a 1 for the Broadcast bit and a 0 for the Use bit of one of its source operands, the bit of the vector corresponding to that physical register is set. The bit vectors from all four clusters are ORed together to form the *Copy Request Vector*. This vector specifies all physical registers requiring a copy instruction. The process of updating this vector is pipelined over two cycles to account for wire delays, and it is later used by the steering logic to insert copy instructions. Assuming all instructions could have at most two source operands, up to 32 bits of this vector could be set each cycle if 16 instructions are issued per cycle. A priority circuit is used to pick up to four physical registers per cluster for which to create copy instructions. The steering logic will then clear the selected bits of this vector and insert copy instructions for the selected physical registers. Copy instructions have issue priority over instructions just fetched from the instruction cache.

Copy instructions are not inserted until at least five cycles after the consumer instructions requiring the re-broadcast have been issued. This 5-cycle delay is due partially to the fact that the steering logic may have already begun to steer instructions that will be issued within the next three cycles, and there is a 2-cycle delay between the clusters' issue logic and the steering logic, which accounts for the delay for updating the Copy Request Vector. Performance is relatively insensitive to this delay since the scenario where copy instructions are needed is rare: only about 1% of retired instructions require copy instructions to be inserted.

The fact that copy instructions are inserted into the window after the instructions which are dependent on them means that deadlock may occur if the following criteria are met: (1) the issue of the copy instruction is stalled because the cluster it needs to be steered to is full, (2) all instructions in that cluster are dependent upon instructions that are dependent upon copy instructions that cannot be issued, hence nothing in the window can be scheduled for execution. To prevent this scenario from ever happening, one entry of each cluster's scheduling window is reserved for copy instructions. Copy instructions are always ready for execution when they are issued and may be removed from the window when they are scheduled. The fraction of copy instructions that are not selected for execution the first cycle they are in the window is less than 1% on average (with the highest fraction being

1.3% in the perl benchmark), so only one dedicated entry is necessary to prevent deadlock.

Table 4.6 shows an example of an inter-cluster broadcast in which a copy instruction is needed. This is similar to the previous example, except that instruction *B* is not issued to Cluster 1 until cycle 3, which is after *A*'s tag was broadcast to that cluster. Because the Use bit of *A*'s BBT entry was not set when the tag was broadcast in Cluster 1, *A*'s data broadcast will be blocked by Cluster 1. When *B* is issued in cycle 3, it reads the Broadcast and Use bits of *A*'s BBT entry. Because *A* had already broadcast its tag but there were no older dependants in Cluster 1, a copy instruction will have to be issued to Cluster 0 to re-broadcast the data. Because it takes five cycles for the copy request to be sent to the issue

			BBT-0		BBT-1	
Cycle	Action		BC	USE	BC	USE
Init	*A* is in Cluster 0.	A		X		
		B				
0	*A* is selected, broadcasts tag to cluster 0.	A	X	X		
		B				
2	*A*'s tag is broadcast to cluster 1. BBT-1[A].Use is clear, so *A*'s tag will not be broadcast 3 cycles later.	A	X	X	X	
		B				
3	*B* is issued to Cluster 1. Read BBT-1[A]. BBT-1[A].BC bit was set and BBT-1[A].Use bit was clear, so insert copy. Clear BBT-1[A].BC. Set Use bits for all of sources and destinations.	A	X	X		X
		B				X
8	Copy instruction issued to Cluster 0.	A	X	X		X
		B				X
9	Copy instruction is scheduled, broadcasts tag to Cluster 0 (no effect).	A	X	X		X
		B				X
11	Copy broadcasts tag in Cluster 1. Set BBT-1[A].BC, Read BBT-1[A].Use, so data will be broadcast. B wakes up.	A	X	X	X	X
		B				X
12	*B* broadcasts its tag in Cluster 1.	A	X	X	X	X
		B			X	X

Table 4.6: Selective Cluster Receipt: inter-cluster forwarding with a copy instruction.

logic and for the copy to be issued, the copy instruction is not issued until cycle 8. It gets scheduled in cycle 9, and broadcasts its tag to Cluster 1 in cycle 11. Finally, *B* broadcasts its tag in cycle 12. If Selective-Cluster Receipt were not used, *B* would have broadcast its tag in cycle 4. Hence there was an 8-cycle penalty for this particular example.

Not only do copy instructions delay the execution of their dependants, but they may take resources away from real instructions performing useful work. They occupy issue ports, possibly causing instructions in the renaming stage to be stalled or steered to an undesired cluster. They will also occupy space in the scheduling window before they are executed, although unlike the models with partitioned register files, they do not remain in the window long because they are already “Ready” when they are placed in the window. They may also prevent a real instruction from being selected for execution as soon as possible, since copy instructions must be selected and access the physical register file like regular instructions. This extra demand on the hardware resources may lower IPC and consume power. However, because copy instructions are only inserted if an instruction’s source operand was steered to a different cluster *and* that operand was already broadcast *and* it was not written to the local physical register file, copy instructions are rarely needed and a small impact on IPC. A summary of the differences in the behavior and requirements of copy instructions in the SCR and PART models is provided in Table 4.7. Power and performance results are shown in Section 4.4.3.

Requirements	SCR	PART
When are they needed?	When a consumer is issued to a cluster other than the producer's cluster after the producer broadcast its tag, and there were no older dependants in that cluster.	whenever a consumer is mapped to a cluster without a valid source without a valid source operand.
Additional RAT mapping?	no	yes
Schedule time?	1 cycle	1 cycle
Bandwidth limitation?	no (full connectivity between clusters)	yes (limited bypass connectivity)
Space in sched window?	yes (usually 1 cycle)	yes (potentially many cycles)
Extra reg file entries?	no	yes

Table 4.7: Differences between copy instructions in the SCR and PART models.

4.4.3 Results for PART and SCR Models

4.4.3.1 Design Space Exploration of PART model

Before comparing models with partitioning against models with replication, the latency, power, and IPC of configurations of the PART model are evaluated. Table 4.8 shows the estimated latency of the scheduling windows and register files when scaling the following parameters (listed left to right in the table): the total number of scheduling window entries in all clusters, the number of scheduling window issue ports (for each cluster), the inter-cluster bypass connectivity (C), and the total number of physical register file entries for all clusters combined. The connectivity is the number of inter-cluster buses, from each cluster, that are sent to each other cluster. Note that the number of register file and window entries in each cluster is the number indicated in the table divided by four, since these resources are partitioned (not replicated) across all four clusters. The purpose of the simulation results is not to find an "optimal" configuration, since no artificial constraint

S.W. Entries	S.W. Wr Ports	C	R.F.	S.W. (ns)	R.F. (ns)
256	4	1	768	0.86	0.45
256	6	1	768	0.90	0.45
256	8	1	768	0.93	0.45
256	12	1	768	1.01	0.45
256	16	1	768	1.08	0.45
256	16	1	768	1.08	0.45
384	16	1	768	1.35	0.45
512	16	1	768	1.61	0.45
640	16	1	768	1.91	0.45
768	16	1	768	1.41	0.45
512	16	1	512	1.61	0.41
512	16	1	768	1.61	0.45
512	16	1	1024	1.61	0.50
768	16	1	768	1.41	0.45
768	16	1	1024	1.41	0.50
512	16	1	768	1.61	0.45
512	16	2	768	1.72	0.51
512	16	3	768	1.83	0.58

Table 4.8: Scheduling Window and Register File Latency estimates, PART model.

is being placed on power consumption. The purpose is to find configurations which can be used to fairly compare this paradigm to other clustered paradigms. Not all possible permutations of these parameters are shown – only those that are used to produce the results shown in this chapter.

Figures 4.11 and 4.12 show the power and IPC of the PART model when varying the number of issue ports per cluster. The power consumption when using 4, 6, 8, 12, and 16 issue ports per cluster is shown relative to the arithmetic mean for the PART configuration with four ports per cluster. All configurations shown use a 256-entry scheduling window and a 768-entry physical register file. As compared to the baseline clustered model (Fig-

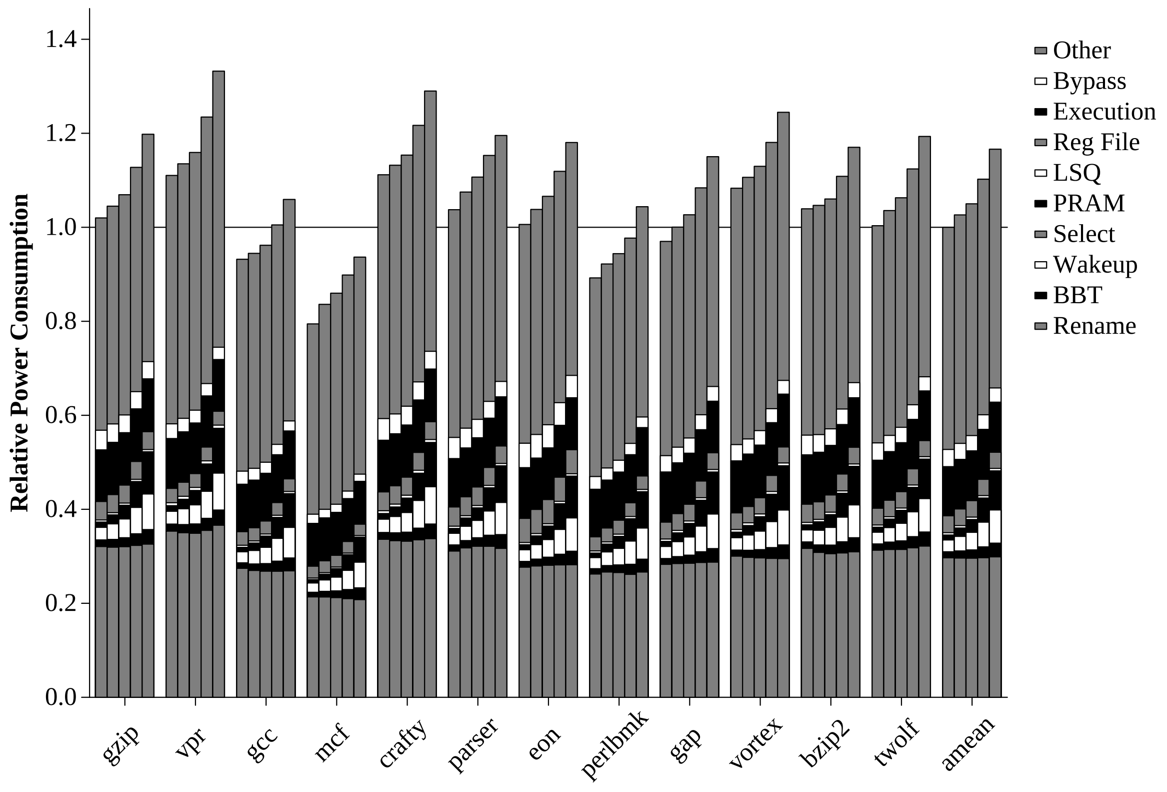


Figure 4.11: PART model Power Consumption while scaling the number of issue ports per cluster : 4, 6, 8, 12, 16 ports.

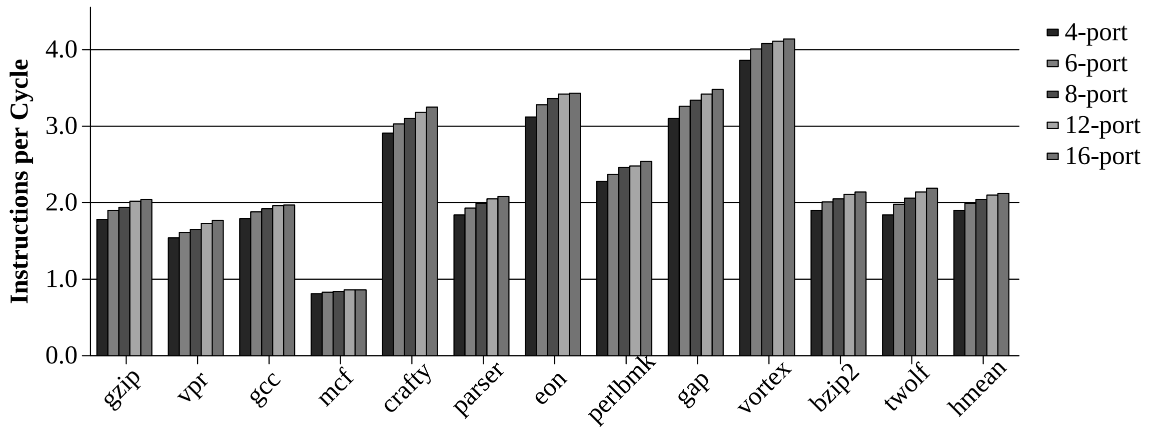


Figure 4.12: PART model IPC while scaling the number of issue ports.

ure 4.7), the IPC of the PART model improves significantly as the number of issue ports is increased. This is because inter-cluster forwarding results in a much larger penalty in the PART model than in the baseline clustered model. In addition to the wire delays of the inter-cluster forwarding paths, the copy instructions must be scheduled. This makes the penalty a minimum of three cycles rather than two. When the number of ports is increased, instructions can more easily be steered to the same cluster as their dependants. Hence there are fewer copy instructions when the number of ports is increased. Additionally, the copy instructions are occupying resources, which further reduces IPC.

Because copy instructions take up scheduling window slots, it is expected that the PART model needs larger scheduling windows than the baseline model. Figures 4.13 and 4.14 show the power and IPC when scaling the number of entries. For all of these configurations, the number of issue ports was held at 16; the size of the physical register file was 786 entries (partitioned among four clusters); and the bypass connectivity was 1. Increasing the size from 192 to 256 improves IPC by 1.5%, and the same improvement is seen when moving to a 384-entry scheduling window. However, the power consumption increases about 5% each time. Hence the most appropriate design would depend on the power budget as well as how potential changes in clock frequency effected both power and performance of other parts of the chip.

Figures 4.15 and 4.16 show the effect on power and IPC when scaling the number of physical register file entries. All configurations use 16 issue ports per cluster. Large scheduling windows were used to prevent full scheduling window stalls from effecting the IPC and to allow the physical register file to be fully utilized. The scheduling window held 512 instructions for the first three configurations, and 768 instructions for the last two. Although copy instructions occupy physical register file entries, performance was not significantly improved by adding more entries. This is partly due to the fact that window utilization is limited by branch prediction accuracy. Averaged over all benchmarks,

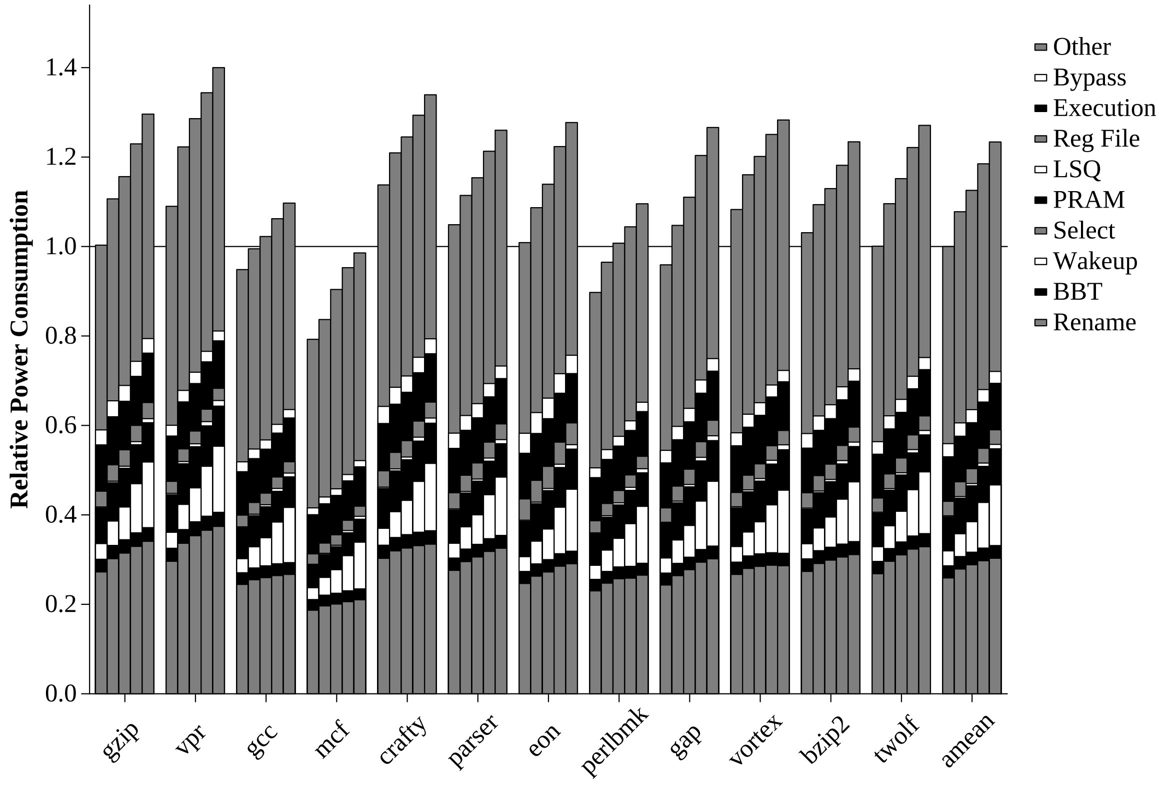


Figure 4.13: PART model relative power consumption when scaling the number of scheduling window entries (128, 192, 256, 384, 512).

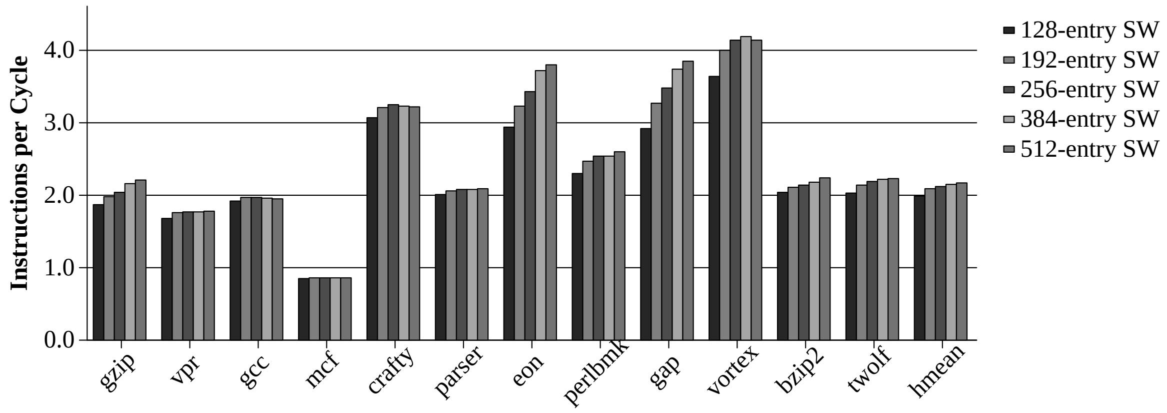


Figure 4.14: PART model IPC when scaling the number of scheduling window entries.

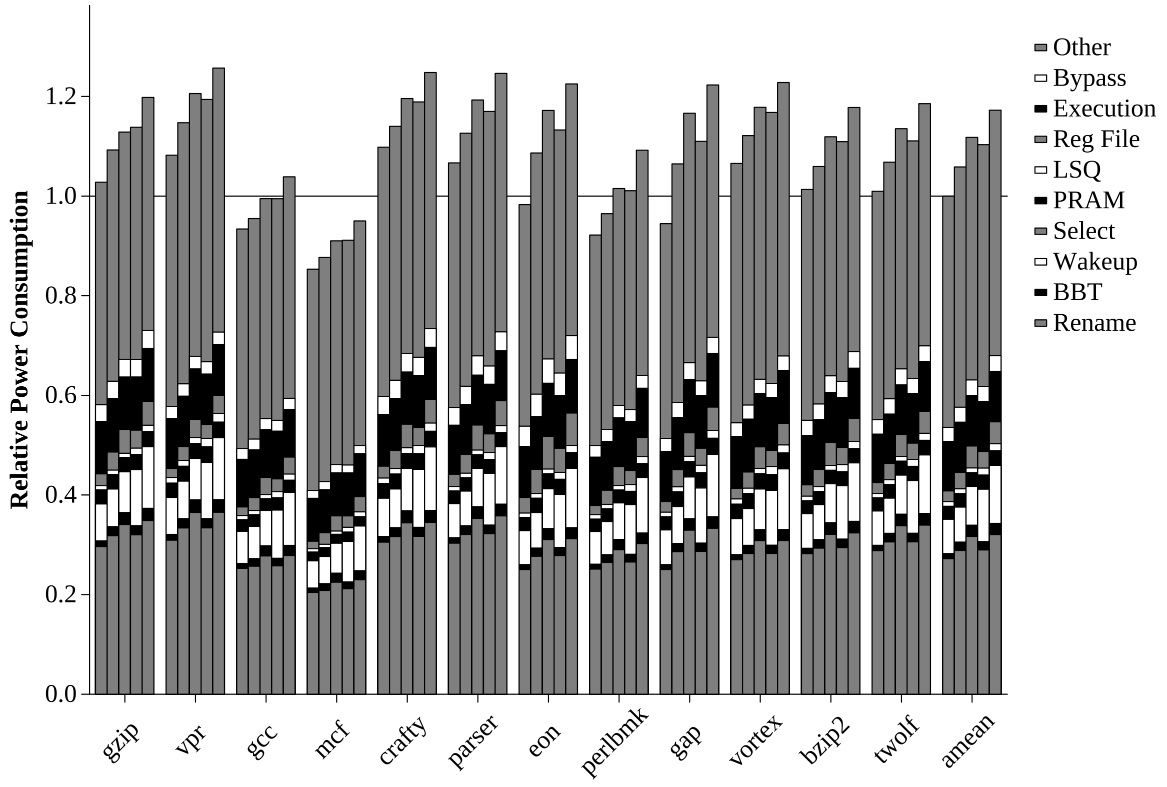


Figure 4.15: PART model relative power consumption when scaling the register file size. SW/RF entries: 512/512, 512/768, 512/1024, 768/768, 768/1024

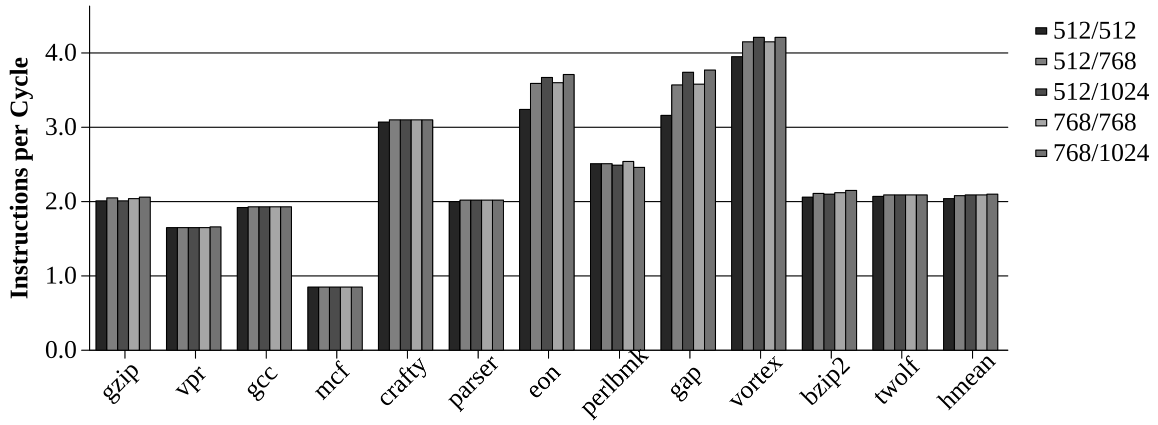


Figure 4.16: PART model IPC when scaling the register file size.

there were an average of 144 instructions between mispredicted branches. (Full window utilization occurs because of the high variance in the number of instructions between mispredicts.) The renaming power changes partially because the number of bits needed to encode each physical register number changes. The power simulator models each RAT entry as containing log(N)+1 bits, where N is the number of physical registers.

Figures 4.17 and 4.18 show the results when changing the bypass connectivity. Increasing the bypass connectivity adds write ports to each register file and tag buses to each scheduling window. Hence the window and register file power increase as the connectivity increases. Increasing the connectivity reduces contention that copy instructions have for the intra-cluster bypasses. However, increasing the connectivity results in little improvement in IPC.

In addition to a limitation on the number of issue ports for each scheduling window, there is a limit on the number of Register Alias Table ports. The number of ports determines the number of instructions that can access the Register Alias Table each cycle. For the baseline clustered model, no more than 16 instructions can access this table per cycle. However with the PART model, copy instructions need to update this table when new physical registers are allocated to existing architectural registers. Figures 4.19 and 4.20 show the effects when the maximum number of instructions that can access the table per cycle is 16, 18, and 20. If enough ports were not available for all instructions within a fetch packet in addition to the copy instructions that had to be inserted, instructions in the fetch packet that did not have access to RAT ports would be stalled in the rename stage. Each increment increases the IPC by 0.5%, although power consumption is increased by about 2% each time.

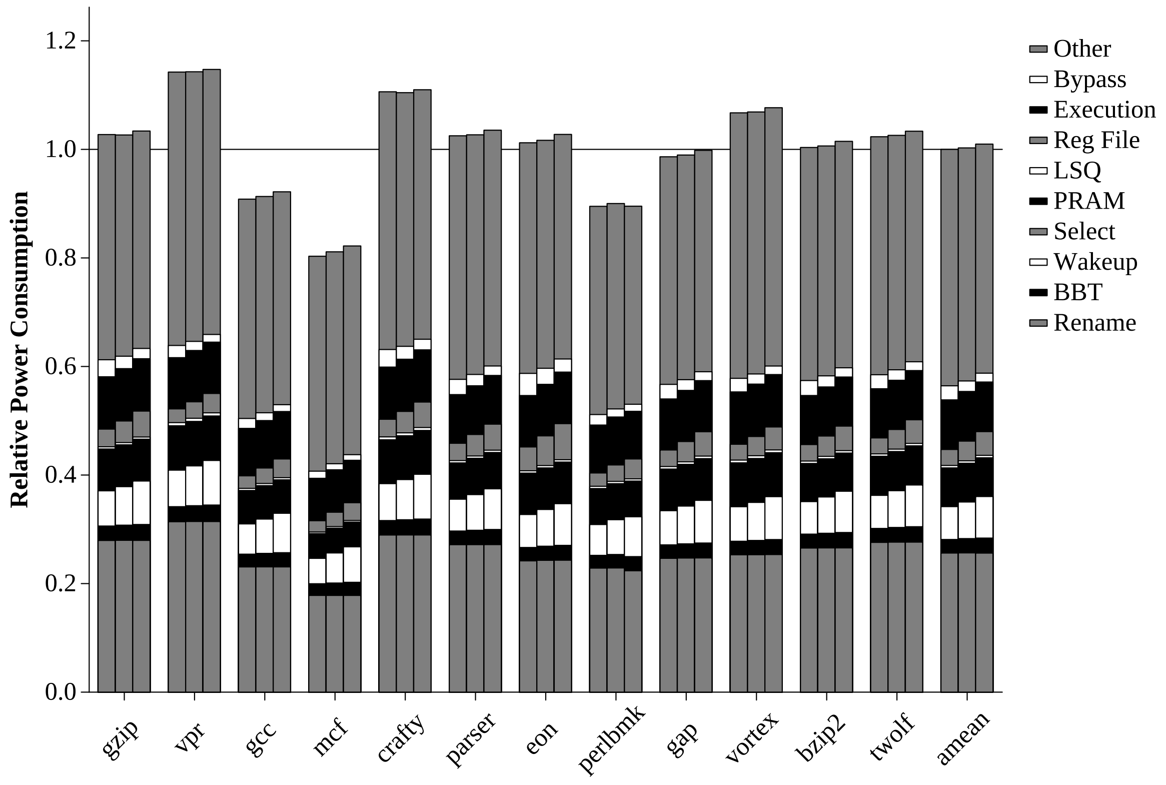


Figure 4.17: PART relative power when scaling the bypass connectivity: 1, 2, 3.

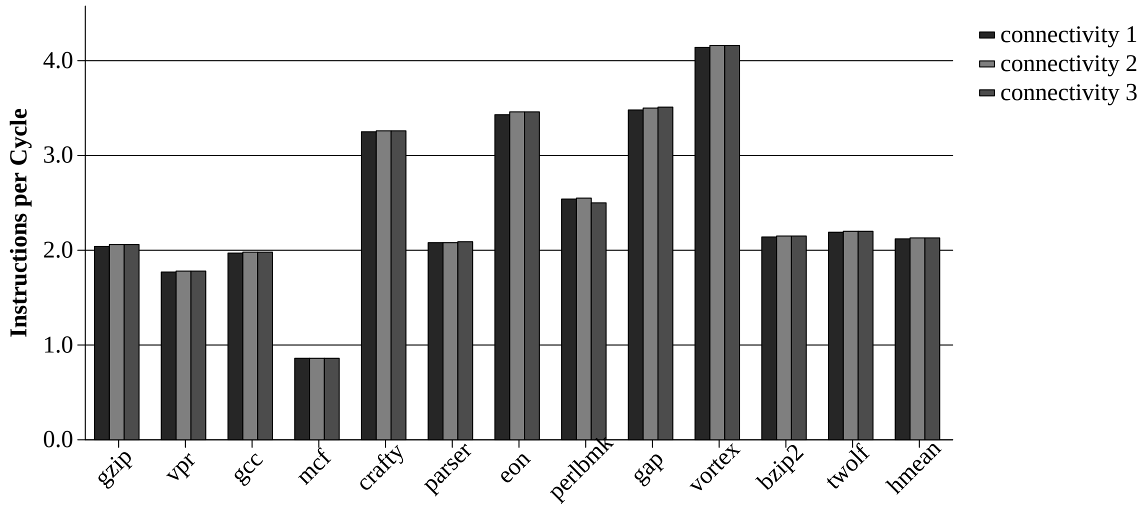


Figure 4.18: PART model IPC when scaling the bypass connectivity.

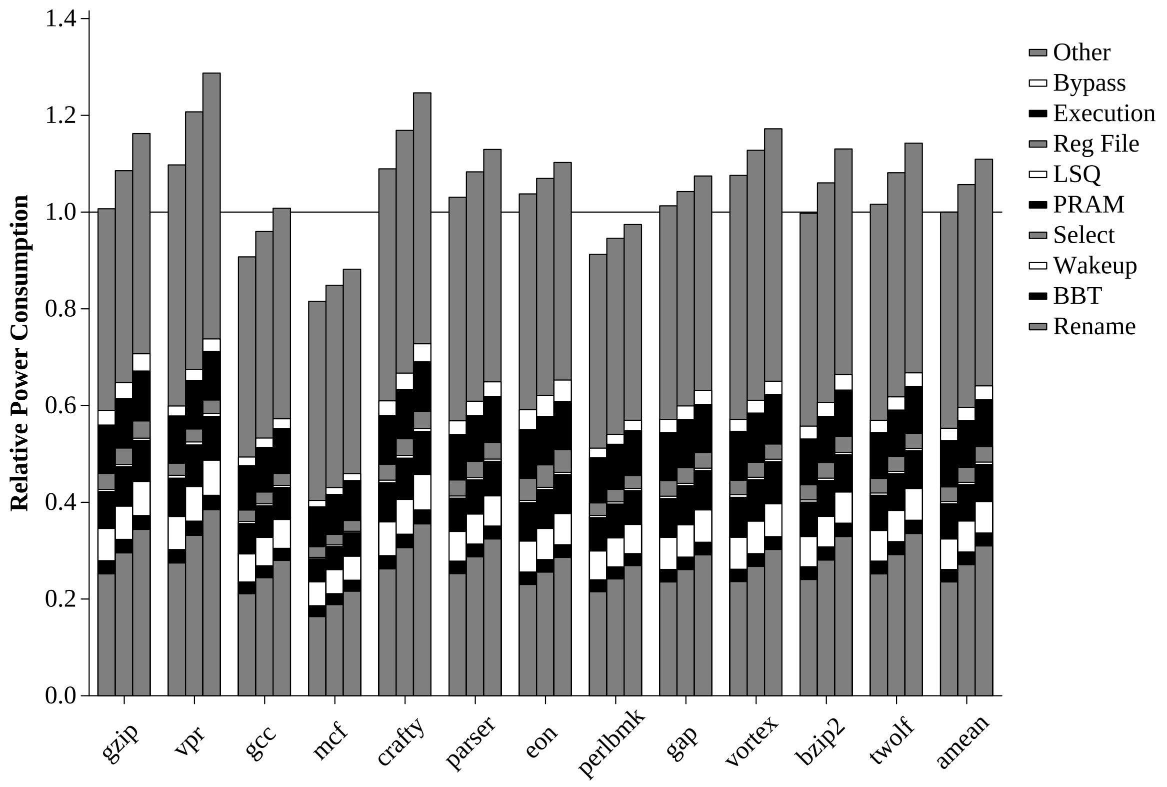


Figure 4.19: PART model relative power consumption when scaling the rename width.

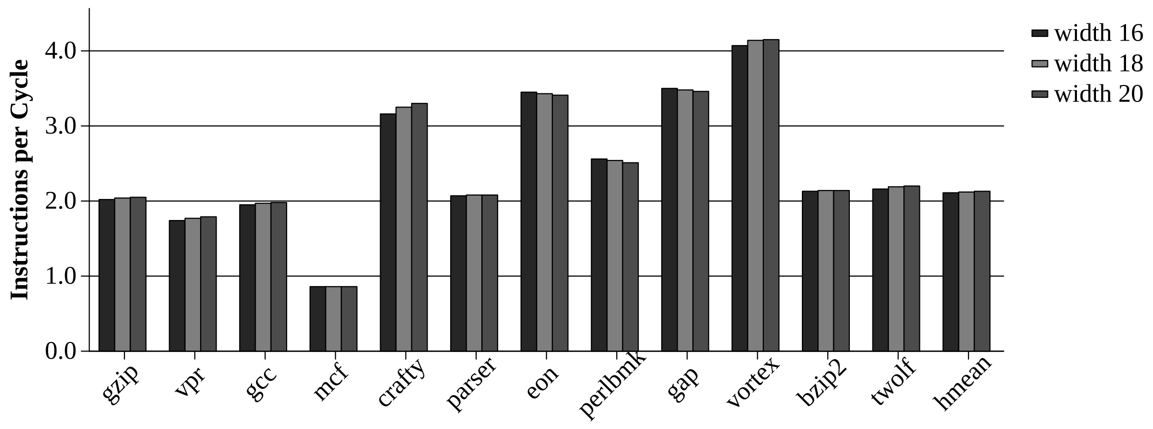


Figure 4.20: PART model IPC when scaling the rename width.

4.4.3.2 Comparing BASE, SCR, and PART Models

In this section, the power and performance of three models are compared: a conventional clustered architecture (BASE), a model with a partitioned register file (PART), and a model implementing Selective Cluster Receipt (SCR). By comparing the latencies in Tables 4.2 and 4.8, we can see that the scheduling windows of the baseline model with 256 entries and six issue ports has about the same (slightly longer) latency as the scheduling window of the PART model with 256 entries and 12 ports. The scheduling window of the baseline model with 256 entries and eight issue ports has about the same latency as the scheduling window of the PART model with 256 entries and 16 issue ports. For both the PART model and the baseline model, using scheduling window entries greater than 256 instructions did not significantly improve performance. Increasing the number of issue ports did, however. Because of this, power and performance results of each configuration with the greater number of ports (eight for the BASE and SCR models and 16 for the PART model) will be shown. The PART models in this section assume a connectivity of 1 since increasing the inter-cluster bypass connectivity did not significantly help performance. The size of the physical register file for the PART model was 768 entries, although it was 512 entries for the BASE and SCR models. A summary of these three models is shown in Table 4.9. The number of scheduling window entries (for all clusters), scheduling window issue ports (for each cluster), and register file write ports (for each cluster) are shown. While the PART model has more register file entries, they are partitioned among

Model	S.W. Entries	S.W. W Ports	R.F. Entries	R.F. W Ports	C	max issue per cycle
BASE	256	8	512	16	n/a	16
SCR-8	256	8	512	16	n/a	16
PART-16	256	16	768	7	1	18

Table 4.9: Summary of three models.

the clusters, whereas the other models all have 512 entries in each cluster. The Connectivity is fixed at the maximum amount (no contention for ports or bypasses) for the BASE and SCR models. With the PART model, up to 18 instructions (some of which may be copy instructions) may be issued per cycle, although no more than 16 may be issued to any given cluster.

Figure 4.21 shows the IPC of these three models. On average, the IPC of the SCR model is less than 0.5% lower than the IPC of the BASE model and 5% higher than that of the PART model. The SCR model performs almost as well as the baseline because copy instructions in this model are rare. The PART model requires significantly more copy instructions, which is why its IPC is lower. Figure 4.22 shows the fraction of instructions that require copy instructions to be inserted. This metric measures only the copy instructions required by correct-path instructions, relative to correct-path instructions, although wrong-path instructions may also generate copy instructions. On average, the PART model requires 12 times as many copies as the SCR model.

Figure 4.23 shows the relative power consumption of these three models. The power consumption of the rename logic is the bottom-most portion of each bar. The power consumption of the rename logic of the PART model is significantly increased for two reasons. The first reason is that the PART model requires multiple mappings of each architectural register, so each entry of the RAT is larger. The second reason is that more write ports for the RAT are needed in the PART model because the copy instructions must also update the RAT. Because rename width is limited to 18 in this model, the RAT has 18 rather than 16 write ports. Additional rename logic is required in this model as well. The SCR model models rename width to 16 because so few copy instructions are needed. While it is not indicated in the figure, 71% of the rename power in the PART model is consumed by the RAT, and 29% is consumed by the register comparators and muxes. In the SCR model, 41% is consumed by the RAT and 59% is consumed by the comparators and muxes.

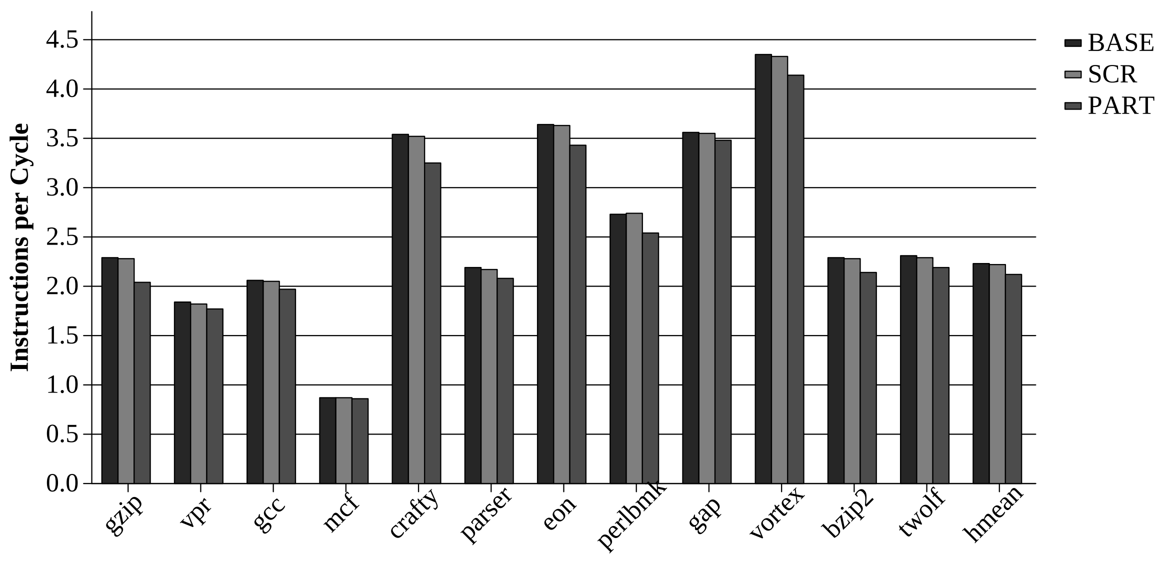


Figure 4.21: IPC of the BASE, SCR, and PART models.

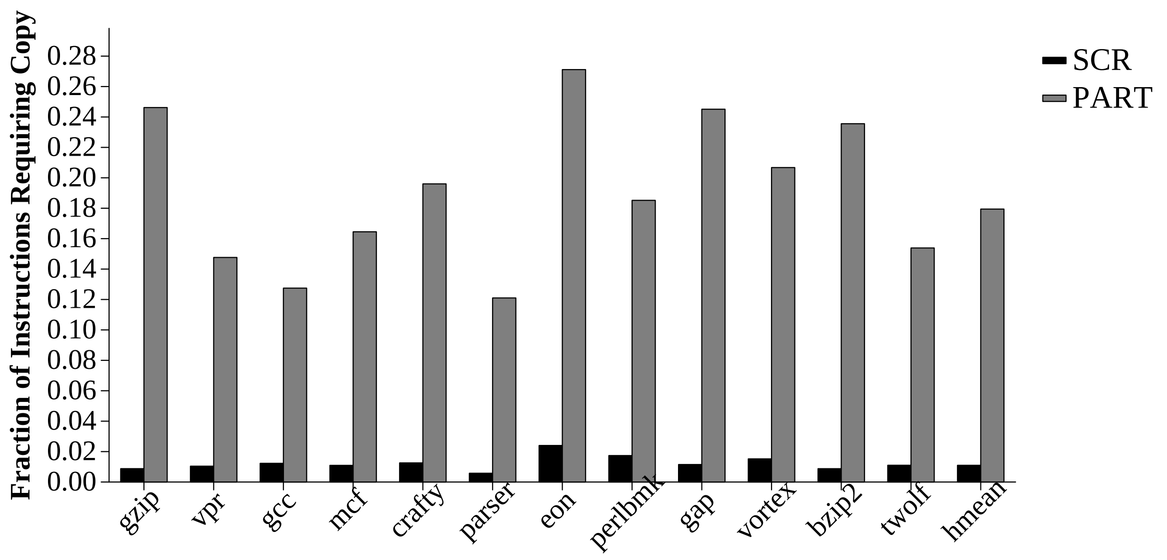


Figure 4.22: Copies per instruction for the SCR and PART models.

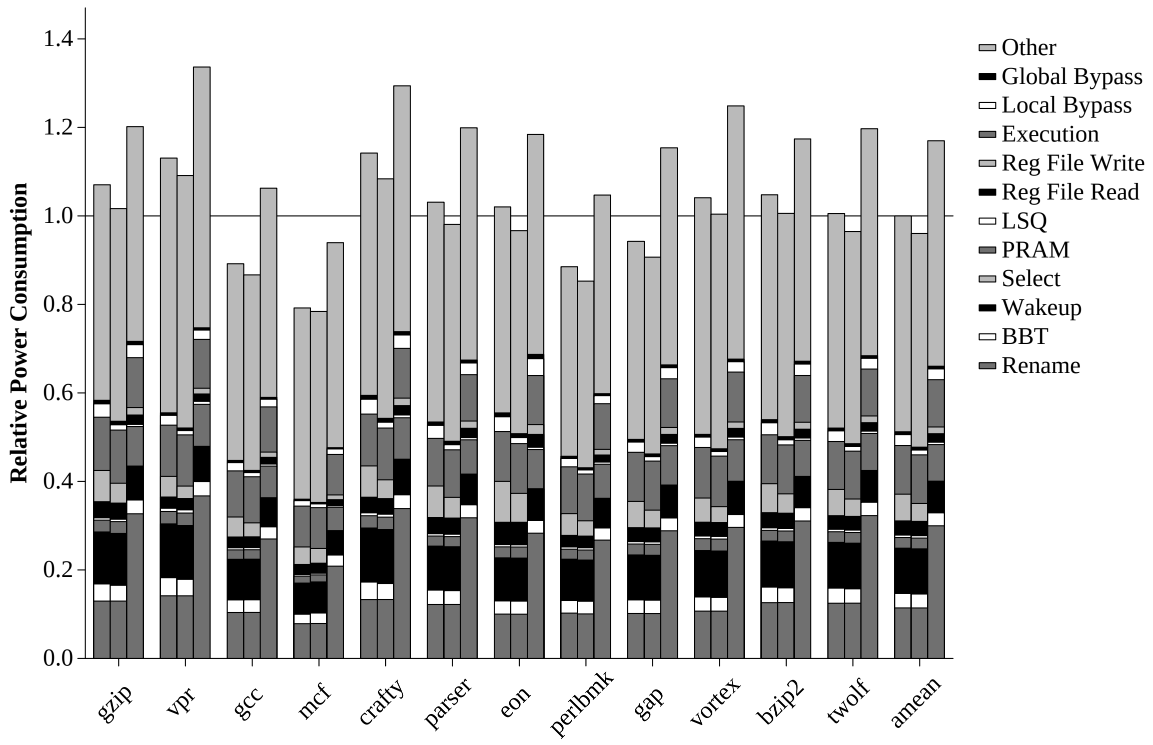


Figure 4.23: Power consumption of the BASE, SCR, and PART models.

The power consumption for the scheduling window ("Wakeup") is the bottom-most black portion of each bar. The PART model has lower power consumption for the scheduling window despite the fact that it has 16 rather than 8 issue ports. This is because its scheduling window has fewer tag buses (7 rather than 16 per cluster) and less CAM matching logic (3584 tag match circuits rather than 8192 tag match circuits). The power consumption of the Select logic is not visible in the graph. The Payload RAM (shown in dark gray above the Wakeup power) for the PART model has higher power consumption because it has more issue ports than the other models. (The number of write ports for the Payload RAM is the same as the number of issue ports for the scheduling window.) The Load-Store queue power is barely visible in most benchmarks, and not visible in others.

The power breakdown in Figure 4.23 separates the register file power into two components: that caused by writes to the register file and that caused by reads. Static power is distributed to the two categories according to the ratio of the areas consumed by write ports and read ports. The register file read power is the second black portion of each bar. The PART model has lower read power because it has smaller register files. The register file write power is above that in light gray. The register file write power for the SCR model is reduced by 33% compared to the BASE model. This is because the number of register file writes is reduced by 66%. The PART model has significantly lower register file power overall. Its register file is much smaller because it has fewer write ports: 7 instead of 16. Additionally, each register file has fewer entries. Table 4.10 shows the average number of clusters in which an on-path architectural register value must be stored. This also represents the number of register file writes per instruction. The PART model requires fewer replications than the SCR model because its scheduling windows have more issue ports. Having more issue ports gives the steering logic the opportunity to steer instructions to their desired cluster. This reduces the number of instructions that are either stalled or sent to a cluster that doesn't contain valid mappings of its source operands. Replications are required when an instruction's cluster does not have valid mappings of the instruction's source operands.

The power consumption of the bypass network is also broken into two parts: the portion of the bypass network internal to a cluster (all four of them), labeled "Local Bypass", and the pipelined buses that run between the the clusters, labeled "Global Bypass." The internal bypass network consists of a bus to the register file and muxes for each of the four functional units in its cluster, whereas the global bypass network is just a collection

Num clusters	1	2	3	4
SCR	65.6%	30.6%	2.4%	0.4%
PART	74.5%	21.2%	1.8%	0.4%

Table 4.10: Fraction of results with given number of cluster broadcasts.

of pipelined wires with low fanout. These two components are shown directly beneath the “Other” category in the figure. As expected, the internal bypass power is reduced for the SCR model due to blocking the intra-cluster broadcasts. The external bypass power is about the same in the SCR and Baseline models because register results are always forwarded on the external pipelined inter-cluster buses.

Overall, the model using Selective Cluster Receipt reduced the power consumption of the baseline model by 5% while having less than 1% impact on IPC. The power consumption for the PART model increased because of the extra work required by the rename logic. Furthermore, it had a 5% reduction in IPC due to the copy instructions.

4.5 Conclusions

Clustering is an effective technique for reducing the latency of the execute-bypass loop. If the execute-bypass loop is too long to fit in one cycle, the IPC is degraded by over 15% as was shown in Chapter 1. However, when clustering, the IPC degradation is less than 10% while offering a potential increase in cycle time or power savings.

This chapter has examined three clustering configurations: (1) conventional clustered models that replicate the register file across all clusters, (2) models that partition the register file, and (3) a model with a replicated register file that implements Selective Cluster Receipt (SCR). SCR was introduced as a way to reduce the number of intra-cluster data broadcasts and register file writes by 66% without requiring very many copy instructions, which lower IPC.

The model with Selective Cluster Receipt performed within 1% of a baseline clustered model with a 5% reduction in power. It had an IPC 5% higher than the model with a partitioned register file while consuming less power. When compared to an unclustered, 16-wide machine, clustering with SCR reduces power consumption by 25% while having half the scheduling latency and a 10% degradation in IPC. Even though the clustered models have a considerably shorter scheduling latency and register file latency, all models assume equal pipeline depths. In implementation, the baseline unclustered model would likely require at least one extra pipeline stage for scheduling, possibly having to implement one of the pipelined scheduling implementations described in Chapter 6 in order to run at the same frequency as the clustered model. Additional pipeline stages for register file access would be needed as well, which may add additional bypass paths. Hence, the actual degradation in IPC of SCR as compared to an unclustered machine could be far less then 10%. The extra pipeline stages were not modeled since the performance degradation would be dependent on the branch prediction accuracy, which is not the subject of this dissertation.

Chapter 5

Steering

Steering is the process of dynamically assigning instructions to clusters. Several previous steering algorithms have been described in the literature [3, 16, 23, 28, 30, 59]. Section 5.1 shows the experimental results from modeling the performance of these algorithms and discusses some of the shortcomings that these algorithms have. Section 5.2 presents new steering algorithms that overcome these shortcomings. Finally, Section 5.3 presents a summary of all of the steering algorithms, including those formerly published.

5.1 Shortcomings of Current Steering Algorithms

There are some microarchitectural features that strongly impact the performance of most steering algorithms that were not considered in previous studies. This section examines those microarchitectural features in the context of previously published steering algorithms. These features include the number of issue ports for each scheduling window partition, and the scalability of these algorithms to wide-issue machines. It also discusses several characteristics of steering algorithms that were not addressed in previous work. Section 5.1.1 will discuss how the number of issue ports for each scheduling window affects the performance of a steering algorithm. Section 5.1.2 will discuss how the occurrence of full window stalls affects the behavior of steering algorithms. Section 5.1.3 explains why there is a critical loop within the steering logic.

5.1.1 Scheduling Window Issue Ports

Recall that *issue* is the process of allocating an instruction in the window, and an *issue port* is a write port for the scheduling window. The number of issue ports to each scheduling window determines the maximum number of instructions that can be steered to a given cluster per cycle. This parameter has not been investigated in previous studies on instruction steering. Most previous algorithms were designed assuming that either the number of ports is equal to the issue width divided by the number of clusters (i.e. the minimum that doesn't restrict issue bandwidth) or else the number of issue ports is equal to the maximum issue bandwidth of the front-end of the pipeline. There are advantages to both approaches, and they will be discussed in this section. Results shown later in this chapter demonstrate that a hybrid of the two approaches (i.e. a weakly restricted amount of bandwidth) works best for many clustering configurations.

By limiting the number of issue ports per cluster, the area of each scheduling window can be reduced. This will reduce both the issue time and the scheduling wakeup and broadcast latency. Table 5.1 shows the relative times required to broadcast a tag and wake up an instruction using scheduling windows with the indicated number of write ports, as compared to the window with four ports. Relative power consumption (when using a dependence-based steering algorithm) was shown in Figure 4.6. The other parameters of the window are the same as the baseline machine: 16 tag broadcasts per cycle, 64 entries per cluster, and two tag matches per entry. IPC must be improved as the number of ports is increased in order to make up for the increases in latency.

Issue ports	4-port	6-port	8-port	12-port	16-port
Relative Wakeup Latency	1	1.04	1.07	1.16	1.24

Table 5.1: Relative Wakeup Latencies when varying the number of issue ports.

The downside of using a limited number of ports per cluster is that it limits the ability to steer an instruction to the same cluster as its source operands. In a worst case scenario, the 16 instructions in one issue packet form a linear chain of dependent instructions. In this situation, it would be beneficial to steer all 16 instructions to the same cluster. If the number of ports per cluster were less than 16, then the dependence chain would be broken across clusters, adding inter-cluster forwarding delays to the execution time of the chain.

5.1.1.1 Modulo Algorithm

Figure 5.1 shows the IPC when assigning clusters using the modulo (MOD) steering algorithm. The modulo steering value for each configuration was equal to the number of issue ports per cluster: the 4-port configuration uses mod-4; the 6-port configuration uses mod-6 and so-on. Using a modulo value equal to the number of ports yielded the highest IPC since this algorithm benefits from using the highest possible modulo value. Since instructions are most likely to be dependent on instructions that are nearby in the instruction stream, the IPC increases as the number of ports increases. These results are very different from those presented by Baniasadi and Moshovosh, but that is mostly because the configuration used in the previous work was an 8-wide machine with only two functional units per cluster. There was more functional unit contention in their machine model.

The IPC is influenced by both the amount of inter-cluster forwarding of source operands and the amount of functional unit contention. Figure 5.2 shows the distribution of the location of each instruction's last available source operand when varying the number of issue ports. The light gray portion at the bottom of each bar represents the fraction of instructions whose last source operand was produced in the same cluster. The dark gray represents the fraction of instructions whose last source operand was forwarded from a different cluster, causing the instruction's execution to be delayed. The black portion of each bar represents those instructions that were ready to execute as soon as they were issued into the window. One can clearly see that as the number of ports and the steering modulo value

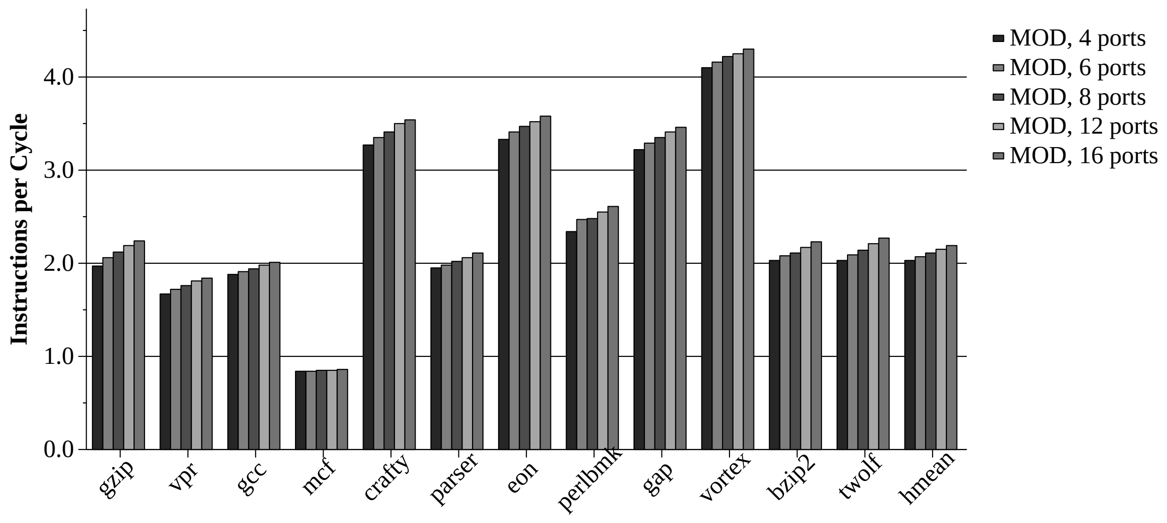


Figure 5.1: IPC using the MOD cluster assignment.

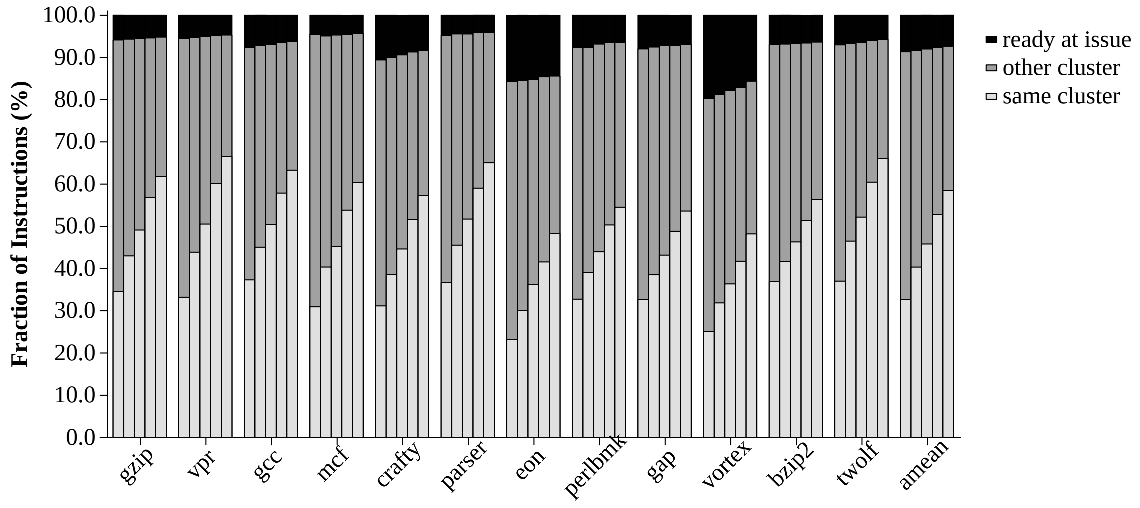


Figure 5.2: Source location distribution using the MOD cluster assignment. From left to right: 4, 6, 8, 12, and 16 issue ports per cluster.

increase, instructions are more likely to be in the same cluster as their source operands. When using only 4 issue ports per cluster, 59% of the instructions received their critical source operand from another cluster. After increasing to 16 ports, only 32% of instructions received their last source operand from another cluster. Figure 5.3 shows fraction of instructions that were delayed because of contention for functional units in their cluster. As the number of ports increases, there is more contention for functional units. Despite the increase in functional unit contention as the number of ports is increased, IPC improved by 6% in the 16-port configuration as compared to the 4-port configuration because of the reduction in the number of critical inter-cluster forwarding delays. However, this increase in IPC came at the cost of a 24% increase in dynamic instruction wakeup latency.

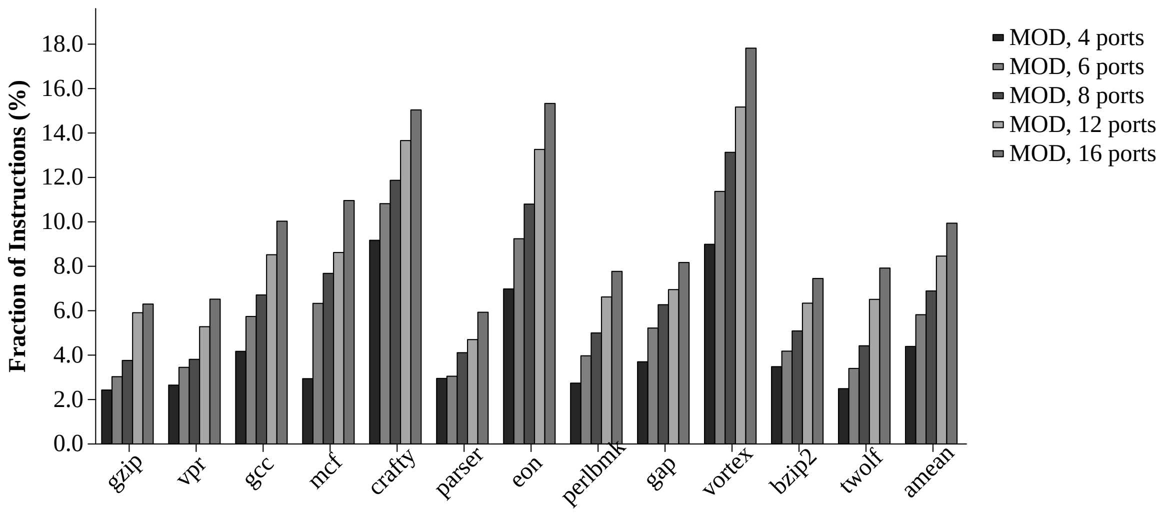


Figure 5.3: MOD: Fraction of instructions for which execution was delayed because of functional unit contention.

5.1.1.2 Dependence-Based Algorithm

Figure 5.4 shows the IPC when using a dependence-based steering algorithm (labeled DEP) which tries to steer instructions to the clusters that contain their register source operands. It works as follows: an instruction's preferred cluster is the cluster containing its producer. If it has more than one register source operand, the cluster of its "left" operand (as determined by the instruction's opcode) is selected. It does not matter if the instruction producing that operand has already executed or even if it has retired. If the desired cluster is full or there are no more available ports for that cluster, then the instruction is assigned to another available cluster. This alternate cluster is chosen by a modulo counter, so it is not necessarily the least loaded cluster. If an instruction has no register source operands, the cluster indicated by the modulo counter is the preferred cluster. The modulo value is equal to the number of ports per cluster. There is a slight improvement in IPC when increasing the number of issue ports per cluster from four to six or eight, but then the IPC levels off or even drops for some benchmarks. This is because this steering algorithm has no load balancing heuristic, and too many instructions get steered to the same cluster.

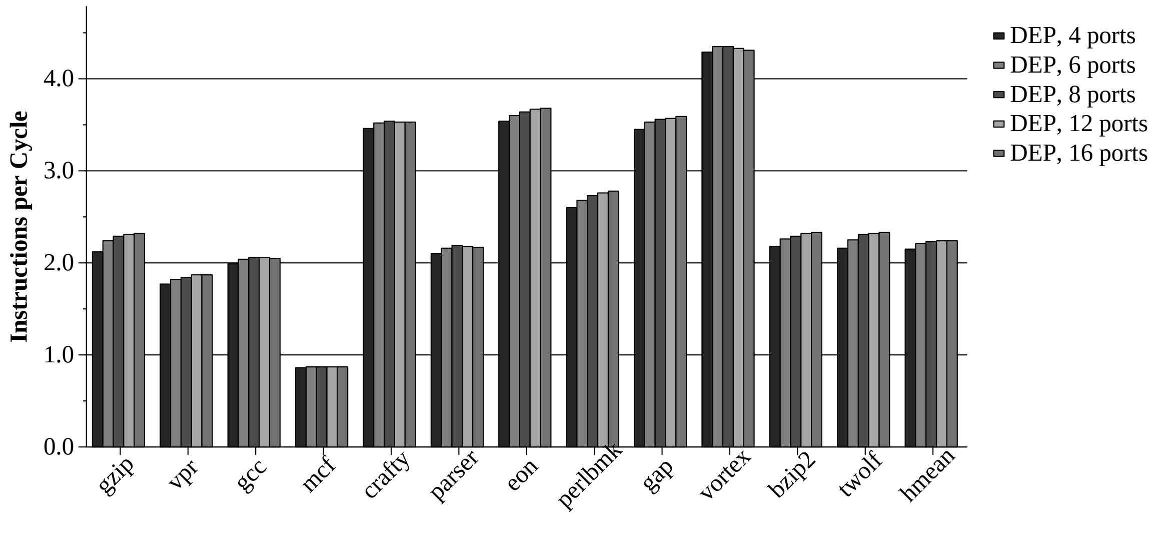


Figure 5.4: IPC using a dependence-based steering algorithm.

5.1.1.3 DCOUNT Algorithm

Figure 5.5 shows the IPC when using the DCOUNT algorithm while varying number of ports per cluster. The DCOUNT is a metric which measures load imbalance, and its value is a function of the number of instructions that have been assigned to each cluster. Each cluster has a counter associated with it that is used to measure load balance. When an instruction is assigned to a given cluster, its counter is incremented by N - 1 (where N is the number of clusters). All other clusters have their counters decremented by 1. Hence, the sum of all counter values is always 0. When the difference between the highest and lowest counter values exceeds a threshold, the instruction is steered to the “least-loaded” cluster. The “least-loaded” cluster is defined as the cluster with the lowest DCOUNT value, not necessarily the cluster with the fewest instructions waiting to execute, although simulations showed that these two metrics resulted in similar IPC. When the threshold is not exceeded, an enhanced version of the DEP algorithm is used. The enhanced dependence-based algorithm incorporates two additional heuristics not used in the DEP algorithm: source operand availability and knowledge of which cluster is least loaded. It works as follows: instruc-

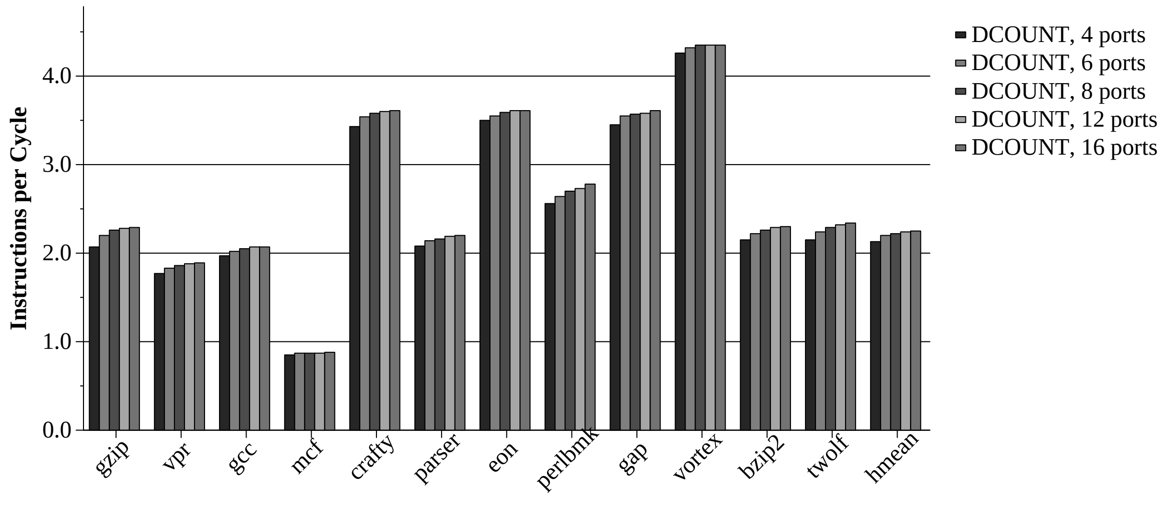


Figure 5.5: IPC using the DCOUNT steering algorithm.

tions are steered to the same cluster as their parent if the parent's result is not available. If the result is available, then the parent's location is not considered by the steering algorithm. If the instruction has more than one source operand that is not available, it is steered toward the least-loaded cluster of the parents' clusters. If all source operands have been produced, the instruction is steered to the least-loaded cluster in which the source operands are available.

Various DCOUNT thresholds in the range of 16 to 128 were evaluated. Regardless of the number of issue ports, the baseline clustered configuration performed best with a very large DCOUNT threshold, meaning the threshold was so large that it had no effect.[1] The DCOUNT algorithm still outperformed the DEP algorithm because of the additional heuristics used by the DCOUNT algorithm. The source operand availability heuristic allows non-critical dependences to be ignored, and the "least-loaded" cluster heuristic improves load balancing. Since the DCOUNT algorithm incorporates these heuristics, the IPC does improve slightly as the number of ports is increased.

The load balancing heuristics used in the DCOUNT algorithm decrease the amount of functional unit contention as compared to the DEP algorithm. However, it also means fewer instructions are steered to the same cluster as their source operands, which means the number of instructions waiting to receive source operands from other clusters increases. Figure 5.6 shows the percentage of instructions whose last source operand comes from another cluster. Except for two configurations simulating the vpr benchmark, the top set of bars are the percentages when using the DCOUNT algorithm, and the bottom set of bars are the percentages when using the DEP algorithm. Overall, for both algorithms, the number of instructions that are delayed decreases as the number of issue ports increases because

[1]Simulations revealed that smaller DCOUNT thresholds were effective on 8-wide machines with four clusters because they had more functional unit contention. Machines with this configuration performed best using a DCOUNT threshold between 16 and 32.

more instructions can be steered to their desired cluster.

Figure 5.7 shows the fraction of instructions whose execution is delayed because of functional unit contention. Except for running the gzip benchmark with four issue ports per cluster, the top set of bars are the percentages when using the DEP algorithm. As the

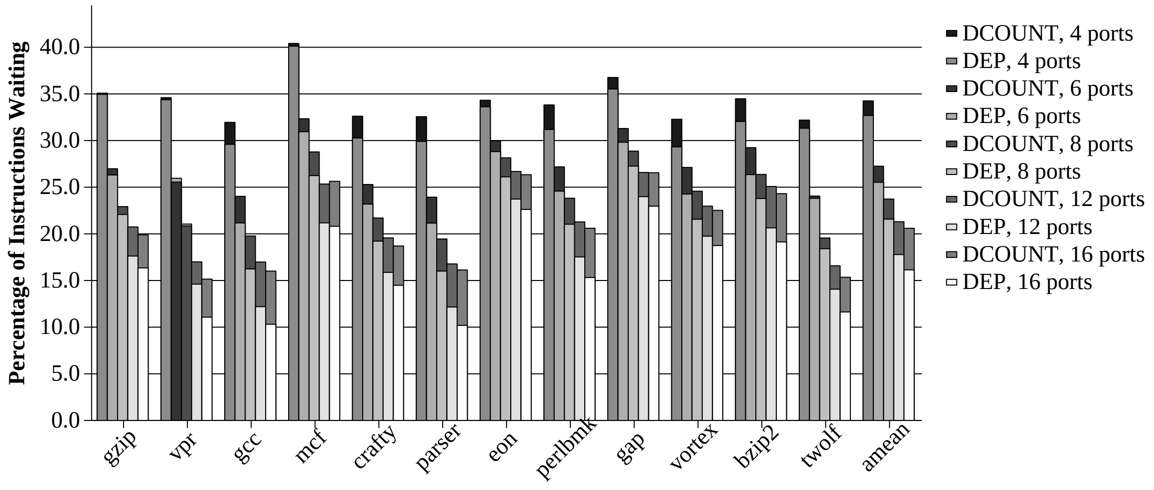


Figure 5.6: Fraction of instructions waiting for the last source operand from another cluster.

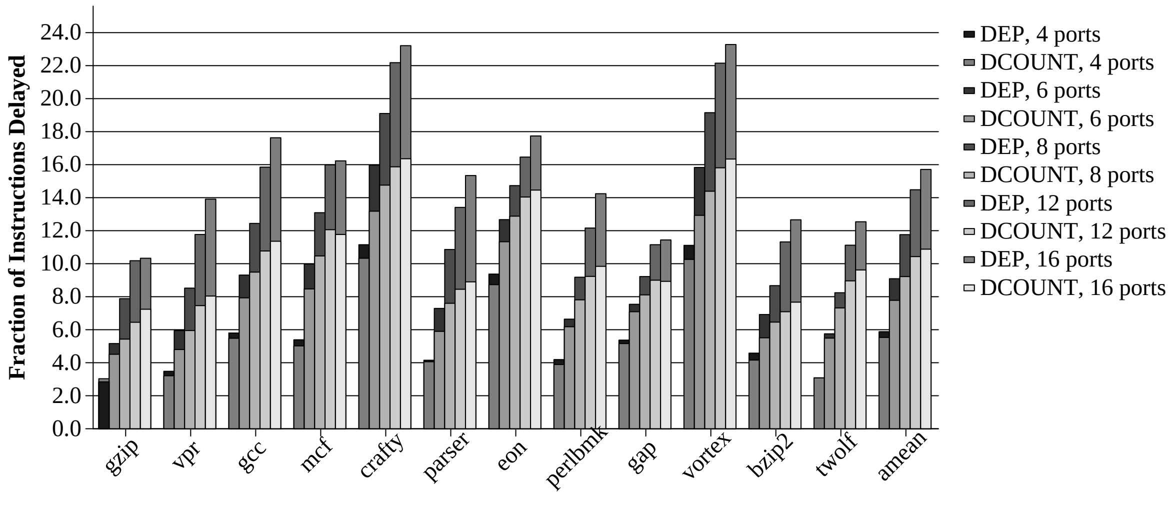


Figure 5.7: Fraction of instructions delayed due to functional unit contention.

number of issue ports is increased, the fraction of instructions that are delayed increases because the workload is not as evenly balanced.

5.1.1.4 FDRT Algorithm

Previous works proposing history-based steering algorithms have assumed each cluster had the minimum number of issue ports (e.g., four ports per cluster in a 16-wide machine with four clusters). There is a reason besides low scheduling latency why this is advantageous. With this limitation, the fill unit can physically reorder the instructions in the trace cache line, and instruction ordering within the trace cache line determines cluster assignment. This means that no crossbar is needed in the front end of the pipeline. For example, the first four instructions are issued to the first cluster; the second four are issued to the second cluster; and so on. This may reduce the latency of the front-end of the pipeline, although this latency may have been overlapped by other operations that occur in the front of the pipeline such as decode and rename.

It was stated earlier that history-based algorithms have no knowledge of the microarchitectural state of the pipeline at the time that instructions are issued to the window. For example, with the trace-cache cluster assignment algorithms, when the fill unit is assigning instructions in a trace packet to clusters, it does not know how full those clusters will be when that trace cache line is fetched in the future. If the number of ports per cluster were greater than four, then there may be a load balancing problem.

To illustrate the effects of increasing the number of issue ports, a trace cache (or rather, the portion of the trace cache holding cluster assignments and enough information to identify basic blocks forming trace cache packets) was added to the baseline model and cluster assignment was performed using the FDRT algorithm [4]. This algorithm was described in Section 2.3.2. The trace cache held entries for 4096 lines (or 65536 instructions), and was indexed similarly to the branch predictor. Each entry needs one or two bits to indicate the cluster assignment, depending on the number of issue ports per cluster and the amount of instruction reordering done by the fill unit. Figure 5.8 shows the performance when using the FDRT algorithm to assign instructions to clusters when scaling the number of issue ports. The IPC actually decreased as the number of ports was increased because load balancing was a problem.

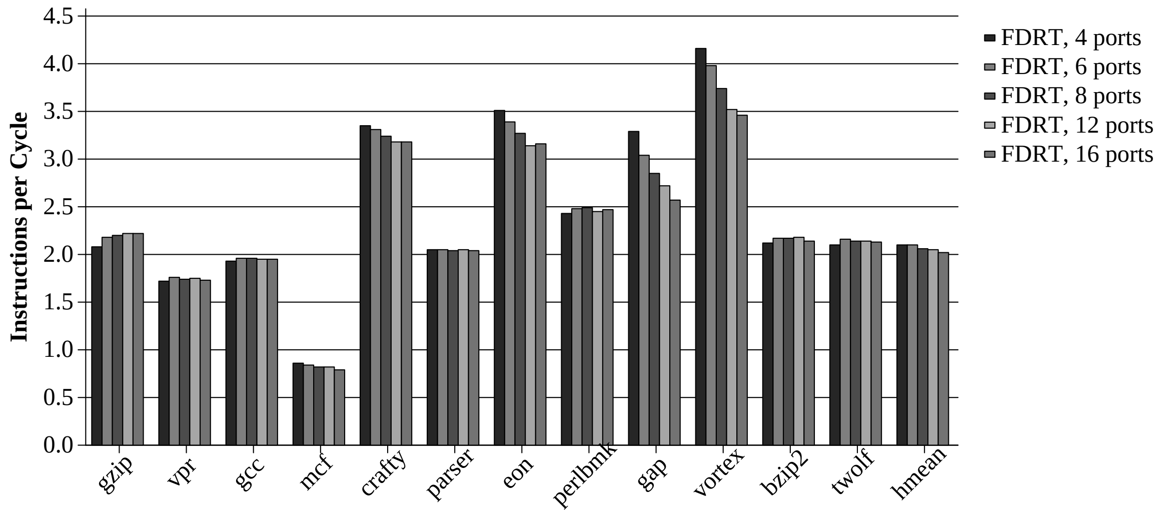


Figure 5.8: IPC using the FDRT history-based cluster assignment.

5.1.1.5 Conclusions

From investigating the effect of the number of issue ports per cluster on IPC for many steering algorithms, we conclude that for most algorithms, fewer than 16 issue ports per cluster are needed to obtain the highest performance. There are three reasons for this. First, the scheduling windows with fewer ports have a lower latency. In some cases it may be advantageous to increase the number of entries and reduce the number of ports in order to implement a window with a specified latency. Second, none of these steering algorithms are optimal. For some algorithms, having fewer ports is beneficial because the limitation acts as a load-balancing heuristic, improving the steering algorithm. Third, it is rarely the case that all last-arriving source operands of all instructions in a fetch packet reside in the same cluster, so there is no need to steer all instructions to the same cluster. Even if all instructions in a fetch packet share the same source operand, they would not all be able to execute in the same cycle because of contention for functional units.

Table 5.2 lists the harmonic means of the benchmarks using the configurations shown in this section. Considering that the FDRT algorithm does not add any latency to the front-end of the pipeline, this algorithm is a good steering candidate when using few ports per cluster. This algorithm cannot be used with many ports per cluster, though. The Modulo steering algorithm is extremely simple in comparison to the other front-end algorithms, but its relative performance is best when using the maximum number of ports per cluster.

Algorithm	4-port	6-port	8-port	12-port	16-port
MOD	2.03	2.07	2.11	2.15	2.19
DEP	2.15	2.21	2.23	2.24	2.24
DCOUNT	2.14	2.20	2.23	2.25	2.27
FDRT	2.10	2.10	2.06	2.05	2.02

Table 5.2: IPC Harmonic Means.

5.1.2 Full windows

Another issue that must be addressed by the steering algorithm is what happens when an instruction's preferred cluster is full. Options include stalling the entire fetch packet, stalling part of the fetch packet (while keeping in-order issue), or steering the instruction to an alternate cluster. Stalling will be investigated in Section 5.2.5. With some steering algorithms, it is quite common for an instruction's desired cluster to be full. Furthermore, when the number of ports per cluster is greater than the minimum, it is more likely for the scheduling windows to become unbalanced. With an unlimited number of issue ports and a pure dependence-based algorithm, all instructions would be steered to the same cluster until that cluster reached full capacity. Most steering algorithms incorporate load-balancing heuristics to prevent this from happening, however.

Figure 5.9 shows the fraction of instructions for which the preferred cluster was unavailable when using FDRT, DCOUNT, and dependence-based steering algorithms, with 4 and 8 write ports per cluster. The results of six configurations are shown. The first three configurations for each benchmark are the configurations with 4 write ports for each issue window, and the last three are the 8-port configurations. Each bar is divided into three parts. The black part shows the fraction of instructions that did not get assigned to their preferred cluster because all ports for that cluster had been claimed by earlier instructions within the same fetch group. Note that this never happens with the FDRT algorithm. With this algorithm, exactly four instructions from a 16-instruction fetch packet are assigned to each cluster. The gray part of each bar is the fraction of instructions whose preferred cluster's is-

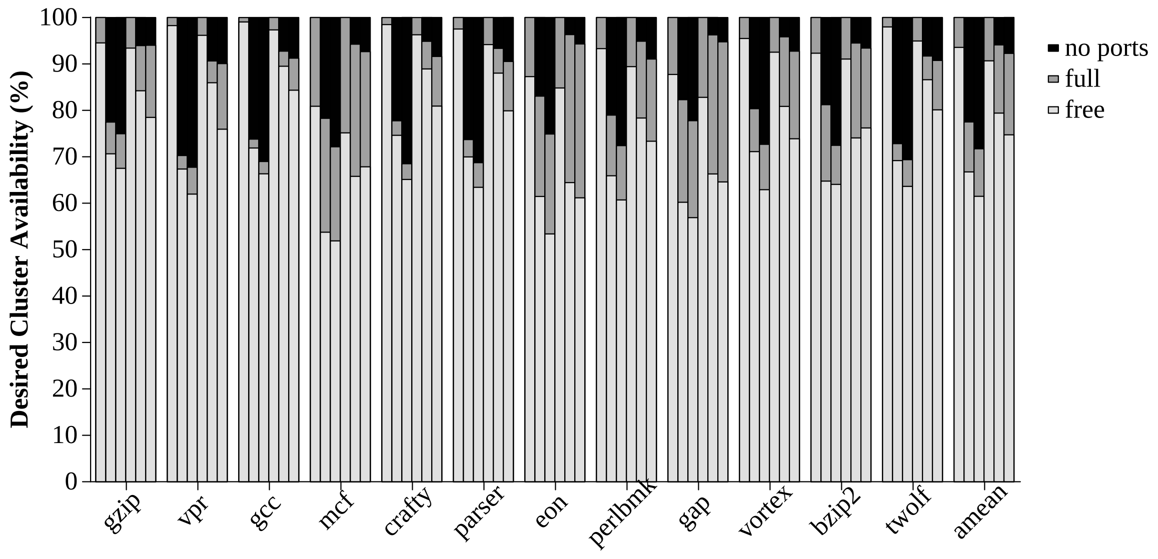


Figure 5.9: Percent of instructions with desired cluster free or unavailable. Six configurations from left to right: FDRT, 4-port; DCOUNT, 4-port; DEP, 4-port; FDRT, 8-port; DCOUNT, 8-port; DEP, 8-port.

sue window was full. For the dependence-based steering algorithm, the average number of instructions that do not get their preferred cluster because all ports have been used by earlier instructions within the same fetch group decreased from 31% to 14% when increasing from 4 to 8 ports per cluster. However, the number of instructions whose preferred cluster was full *increased* from 7.4% to 13.8%. This is because the dependence-based algorithm does not use an explicit load-balancing heuristic. The limitation of the number of ports acts as a natural load-balancing heuristic for this algorithm. Because the DCOUNT algorithm does have some load-balancing heuristics, more instructions can be steered to their preferred cluster. It should be noted that the *preferred* cluster, as determined by the steering algorithm, is not always the best cluster in the long run. For example, for the 8-port configurations, 85% of instructions got their preferred cluster with the DCOUNT algorithm while only 76.4% of instructions got their preferred cluster with the dependence-based algorithm, yet these two configurations had about the same average IPC.

5.1.3 Critical Loops Within the Steering Logic

Dependence-based steering algorithms are inherently serial. Given an instruction B that depends on an instruction A, B will be steered to A's cluster using a pure dependence-based algorithm. This means that the cluster assignment for B cannot be made until the cluster assignment for A has been made. Note that this is not true for register renaming, in which all instructions in an issue packet can be renamed in parallel. With register renaming, instruction B's *destination* physical register is independent of A's destination register. All destination registers can hence be assigned in parallel[2]. The process of identifying the source operands is described in Section 3.1.2. Pipelining the steering logic does not alleviate the serialization problem. Dependence-based steering algorithms create a serialization

[2]This assumes that physical register numbers are independent of cluster assignment, as is the case for the baseline clustered machine.

bottleneck in an otherwise instruction-parallel pipeline.

The critical loop of these algorithms is the process of assigning one instruction to a cluster given the cluster assignments of all previous instructions. A conceptual view of three such loops is shown in Figure 5.10. Each box represents the logic needed to implement the steering heuristic for steering one instruction. The RAT contains cluster assignments for all architectural registers produced by instructions in previous fetch packets. The inputs on the left side of each box are the locations of the source operands, which may come either from the RAT, or a bypass from an earlier instruction in the same fetch packet. The inputs on the top are the cluster assignments for all older instructions in the fetch packet. For a pure dependence-based algorithm that assigns instructions to the same location as its “left” source operand (as determined by the binary instruction format) and does not consider the number of instructions already assigned to each cluster or the fullness of each cluster, the critical path through this logic is just the path through all muxes. However, if metrics such as cluster fullness or the number of available ports left over after assigning earlier instructions are needed by the steering algorithm, the critical path may be much longer. These metrics must be considered by the steering algorithm unless issue

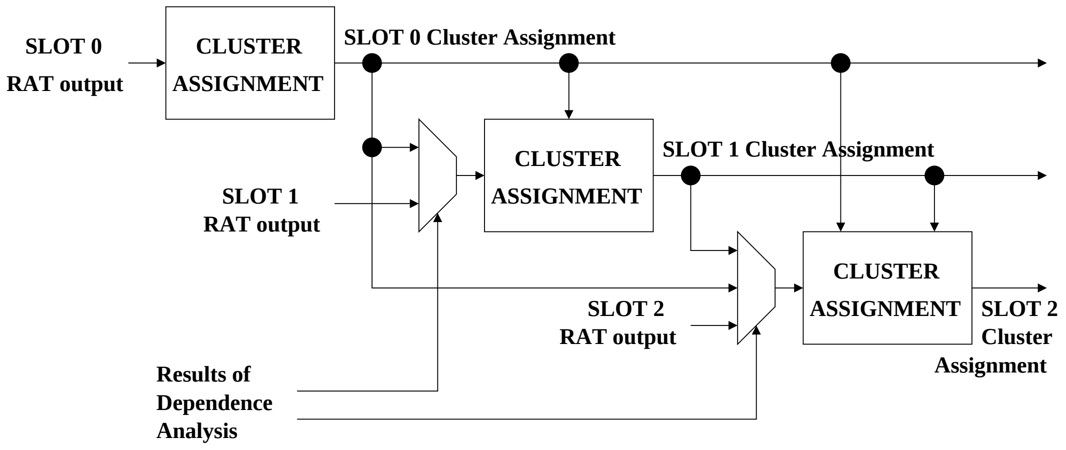


Figure 5.10: Serialized Steering Logic.

is stalled when a resource is unavailable. Scalable dependence-based algorithms will be discussed in Section 5.2.2.

5.2 Steering Improvements

5.2.1 Virtual Cluster Assignments

The history-based algorithms have the advantage that they remove the steering algorithm from the front end of the processor pipeline, which would probably reduce the branch misprediction penalty. Additionally, they may remove the crossbar that is needed to route instructions to clusters when the number of issue ports per cluster is the minimum (i.e. the total issue bandwidth is equal to the fetch bandwidth). Increasing the number of ports has been shown to improve IPC for previous steering algorithms. However, when the number of issue ports per cluster is greater than the minimum, load balancing becomes a problem for history-bsed algorithms as was demonstrated in Figure 5.8. The problem is that for several consecutive fetch packets, a majority of the instructions get assigned to the same cluster because the algorithms utilize information about the dependence graph. Over time, some clusters may become full while others are relatively empty.

To correct the load balancing problem, we treat the cluster assignment assigned by the fill-unit as a *virtual* assignment. That assignment is then renamed in the front-end of the pipeline to a physical cluster. The logic required to rename the clusters is simple and does not increase the length of the front-end of the pipeline because it is placed in parallel with other longer-latency logic such as decode logic. Cluster renaming does not require the complex logic needed for cluster assignment such as data dependence analysis and counting the number of instructions from the checkpoint assigned to a given cluster. That logic can be kept in the fill-unit. On the front-end of the pipeline, only a simple virtual to physical mapping is needed.

Virtual cluster assignments were added to the best-performing previously published history-based algorithm: FDRT [4]. This algorithm was described in Section 2.3.2. When using cluster renaming in conjunction with the FDRT algorithm in the fill-unit, the FDRT algorithm is not modified. The front-end of the pipeline contains a renamer that creates a 1-to-1 mapping from virtual to physical cluster numbers. The mapping is modified when one of the clusters desired by an instruction is full. The mapping changes as follows: the mappings of the most and least-loaded clusters are swapped, and the mappings of the other two clusters are swapped. Figure 5.11 shows the speedups achieved over the original FDRT algorithm when clusters are remapped dynamically. Speedups of the 6, 8, 12 and 16-port configurations with remapping are compared to the FDRT algorithm using only 4 issue ports per cluster, since that configuration had the highest IPC when using the original FDRT algorithm. Overall, using virtual cluster assignments with the FDRT algorithm can

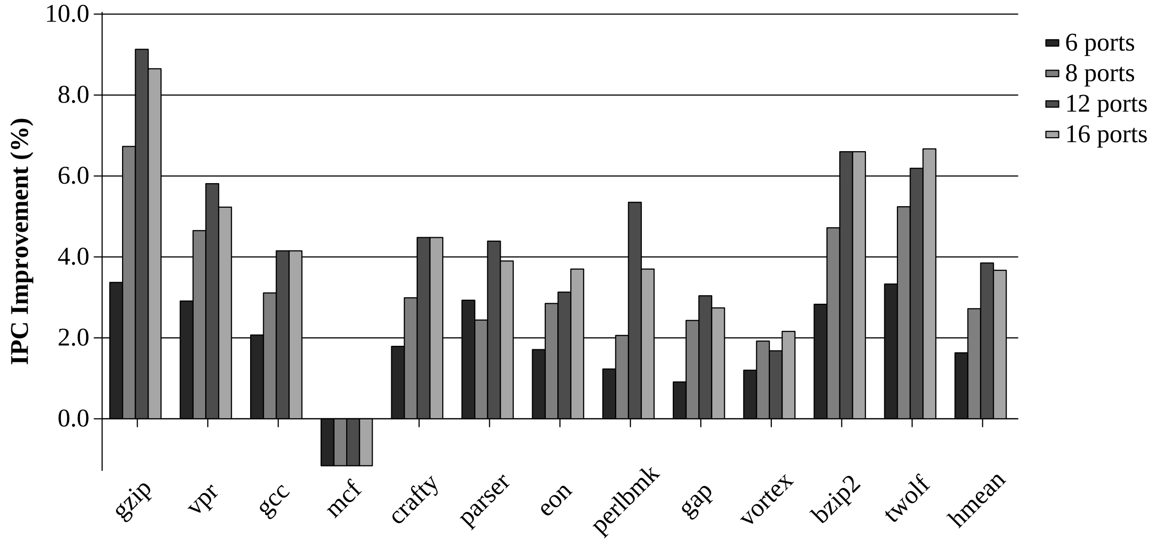


Figure 5.11: Speedups of FDRT with cluster remapping (FDRT+VC) over original FDRT with 4 ports.

allow up to a 4% improvement in IPC, on average.[3]

5.2.2 Scalable Dependence-Based Algorithms

The serialization created by the critical steering loop can be avoided by using dependence chain depths to limit the number of instructions in a fetch packet steered per cycle. A similar technique was used by Ernst et al. [22] to reduce the critical path of instruction prescheduling logic [44]. With their technique, dependence chains were limited to a depth of 2 in a 4-wide pipeline. If a group of instructions had a chain depth greater than 2, then instruction issue was stalled. They note that this has little performance degradation because most of the instructions that are stalled are not ready for execution anyway. Figure 5.12 shows the fraction of retired instructions at a given chain depth. If an instruction has no source operands produced by older instructions within the same fetch packet, then it has a

[3]The mcf benchmark performed worse with more than 4 issue ports per cluster because the contention for data cache ports was increased. Each cluster has access to only a subset of the L1 data cache ports, and loads that were steered to the same cluster contended for those ports.

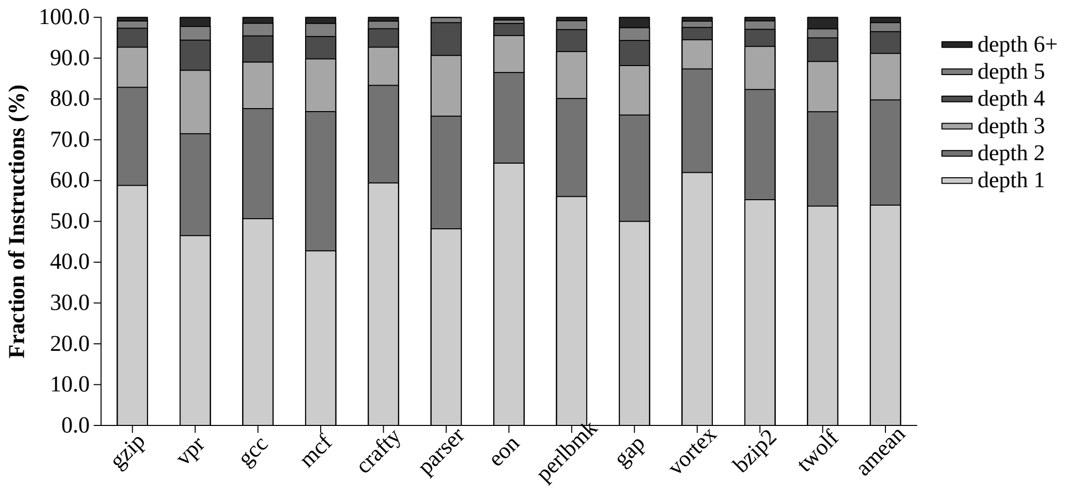


Figure 5.12: Chain depths of retired instructions.

chain depth of 1. Otherwise, an instruction takes the maximum chain depth of all parents residing in the same fetch packet and adds 1. For example, in gzip, 59% of instructions did not have source operands produced by older instructions in the same fetch packet. 24% of instructions had a parent in the same fetch packet, but no grandparents within the same fetch packet. On average for all benchmarks, 20% of instructions had a chain depth of 3 or greater, meaning at least one of their grandparents was in the same fetch packet.

Because very few instructions have a large chain depth, dependence-based algorithms can stall issue when an instruction's chain depth exceeds a certain limit without a large performance degradation. Figure 5.13 shows the reduction in IPC of the DEP algorithm when the dependence chains were limited to depths of 2, 3, and 4. For example, with a maximum chain depth of 3, if an instruction were in the same fetch packet as any of its great-grandparents, then its issue (along with the issue of all younger instructions) would be stalled until the following cycle. These slowdowns are relative to a 16-wide pipeline with four clusters using a dependence-based algorithm that tries to send instructions to the same cluster as their "left" source operand. As long as the cluster assignment logic allows

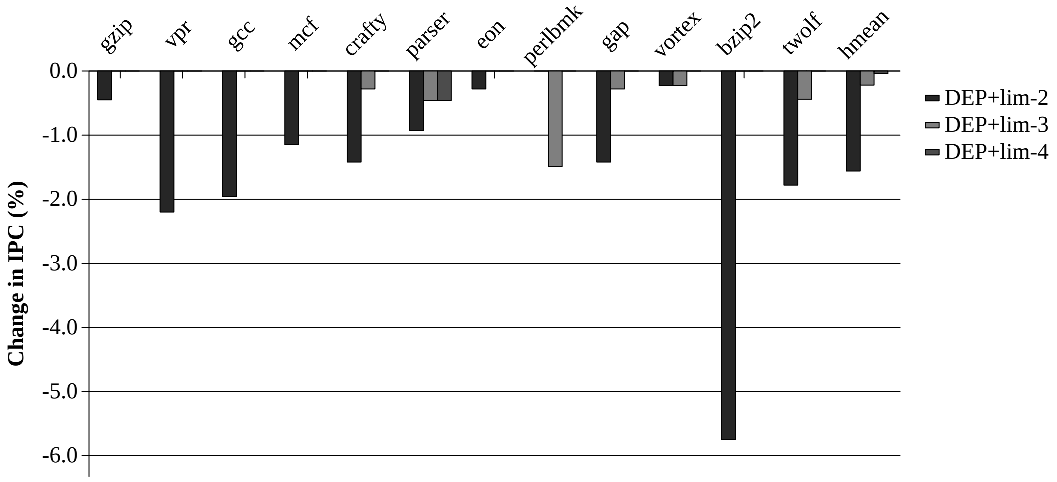


Figure 5.13: Limiting dependence chain depths to 2, 3, and 4 (DEP+`lim`-N vs. DEP).

dependence chain depths of at least 3, IPC is not significantly degraded.[4]

5.2.3 Out-of-Order Steering

Cluster assignment for all previous front-end algorithms was done in program order one instruction at a time (still several per cycle in a superscalar processor). However, as shown in Figure 4.3, many instructions have no cluster preference. To exploit this fact, we could assign instructions to clusters out of program order so that instructions with no cluster preference don't get assigned to a cluster that is in high demand, preventing younger instructions from being steered to that cluster. Note that the process of issuing instructions into the window is still performed in program order.

Figure 5.14 shows the performance of a limited out-of-order steering algorithm, called 2PASS, compared to the DCOUNT and DEP algorithms when using four and six ports per cluster in 4-cluster configurations. The algorithm used here is quite simple: it is a two-pass dependence-based algorithm. Two passes are made over each fetch packet of instructions. On the first pass, only instructions that are waiting for a source operand to be produced are assigned to a cluster. Those instructions are assigned to clusters using the DEP algorithm. The second pass assigns all remaining instructions to clusters. An instruction cannot be assigned to a cluster during the first pass unless its parent was assigned to a cluster during the first pass. The second pass does not use dependency-graph information; instructions are simply assigned to the least-loaded cluster. This algorithm uses the same heuristics as the DCOUNT algorithm, but has the advantage of assigning instructions out-of-order. This allows more instructions to be placed in the same cluster as their source

[4]The large performance degradation in bzip2 with a chain depth limit of 2 is due to the fact that frequently a chain of 3 or more instructions ends in a mispredicted branch. The execution of the mispredicted branch gets delayed, thus increasing the total number of cycles spent fetching the wrong path. The instruction window utilization was reduced by 8% for this confi guration, while the number of cycles spent fetching down the wrong path increased by 34%.

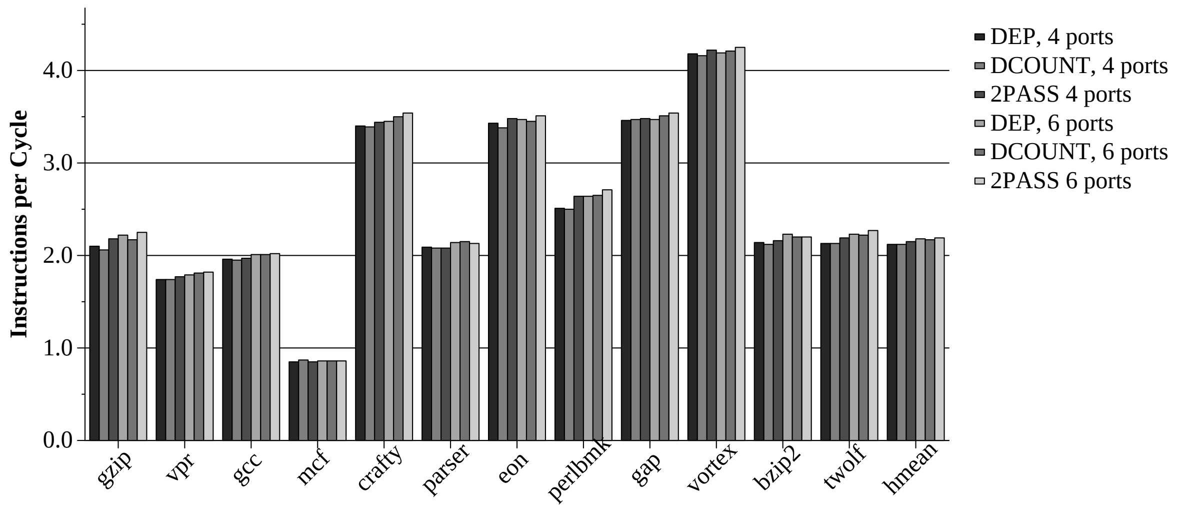


Figure 5.14: IPC for DEP, DCOUNT, and a 2-pass out-of-order algorithm with 4 and 6 ports per cluster.

operands, particularly when the number of issue ports for each cluster is small. When compared to the DCOUNT algorithm, it improves IPC by 1.5% when using 4 issue ports per cluster, and by 1% with 6 ports. Because this algorithm is most beneficial when using few ports per cluster, results with more issue ports were not shown.

A second out-of-order steering algorithm, called a *tree-based* steering algorithm, combines out-of-order steering with virtual cluster assignments. This steering algorithm has three steps: the first two steps assign instructions to virtual clusters; the third step maps the virtual clusters to physical clusters. In the first step, instructions in a fetch packet are assigned to groups using a dependence-based heuristic. If an instruction is the first one in the fetch packet or the instruction's source operands were produced by instructions in previous fetch packets, then the instruction is allocated a new group identifier. If an instruction's source operand was produced within the same fetch packet, then the instruction is assigned to the same virtual cluster as its source operand, so long as the number of instructions in that virtual cluster does not exceed the number of scheduling window write ports. By default, as with the other dependence-based algorithms, the “left” source operand is used, unless that operand was produced by an instruction in a different fetch packet. Because the dependency graph formed by the instructions in a virtual cluster form a tree, the instructions in a given virtual cluster will henceforth be called a *trees*. There may be more or fewer trees than physical clusters. If all 16 instructions in a fetch packet are independent, then they will form 16 separate trees.

Figure 5.15 shows an example of the dependency graph of the instructions in a fetch packet from the eon benchmark. Dependences to instructions in earlier fetch groups are not shown since they are not used by the algorithm. The numbers in each node indicate the program order of the instructions, and the dotted lines mark the groups of instructions that form trees. Instructions 0, 1, 3, 5, 10, and 14 have no source operands generated by older instructions in the same packet, so they each start a new tree. Hence, there are a total of six trees for this fetch packet. Note that instruction 7 is placed in the tree with instruction 2 because it uses the dependence indicated by its left source operand, leaving instruction 5 to be in a tree of its own (there is no backwards propagation of the dependence graph).

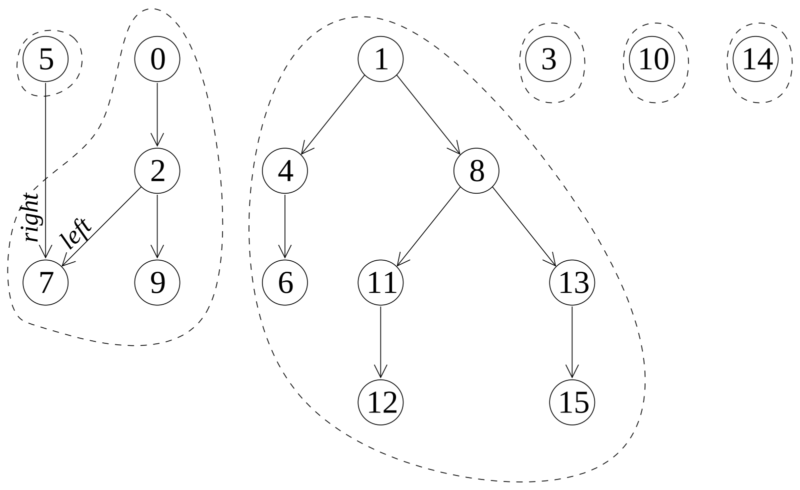


Figure 5.15: Dependency graph and trees for one fetch packet.

The second step is to map trees to virtual clusters. For this algorithm, the number of virtual clusters is the same as the number of physical clusters: four in the baseline clustered configuration. Several heuristics were evaluated to group the trees into clusters. Ideally, all instructions in a given tree will be mapped to the same cluster, but this does not always happen due to a limit on the number of write ports in each cluster. The most complex heuristic sorts the trees according to the number of instructions they contain, and minimizes the number of cuts while mapping the trees to clusters. A simpler algorithm was implemented as follows: first, any trees larger than N instructions (where N is the number of issue ports per cluster) are broken. The N oldest instructions form a sub-tree, then the next N form another tree, and so on. For example, in Figure 5.15, instructions 1, 4, 6, and 8 form a tree, and instructions 11, 12, 13, and 15 form a tree, assuming four issue ports per cluster. Second, the trees are assigned to virtual clusters from largest to smallest. For the example above, the three trees each with four instructions are assigned a virtual cluster, and the remaining four trees with one instruction are assigned the last virtual cluster. While this is complex because the trees must be sorted by size, the fill unit can perform this work (i.e. forming trees and virtual cluster assignments) in parallel for each fetch packet, and the information is stored in a trace cache.

The third step assigns virtual clusters to physical clusters according to dependences: a virtual cluster is mapped to the physical cluster which has the greatest number of live-ins for the instructions in the virtual cluster. Figure 5.16 shows the improvements of using this algorithm over the dependence-based algorithm. This algorithm was beneficial when using fewer ports per cluster. However, IPC was lower when using many issue ports per cluster. This is because the algorithm is effective at reducing the number of inter-cluster communications, but there was a larger amount of contention for execution resources with this algorithm.

Out-of-order windows spanning more than one fetch group are not considered be-

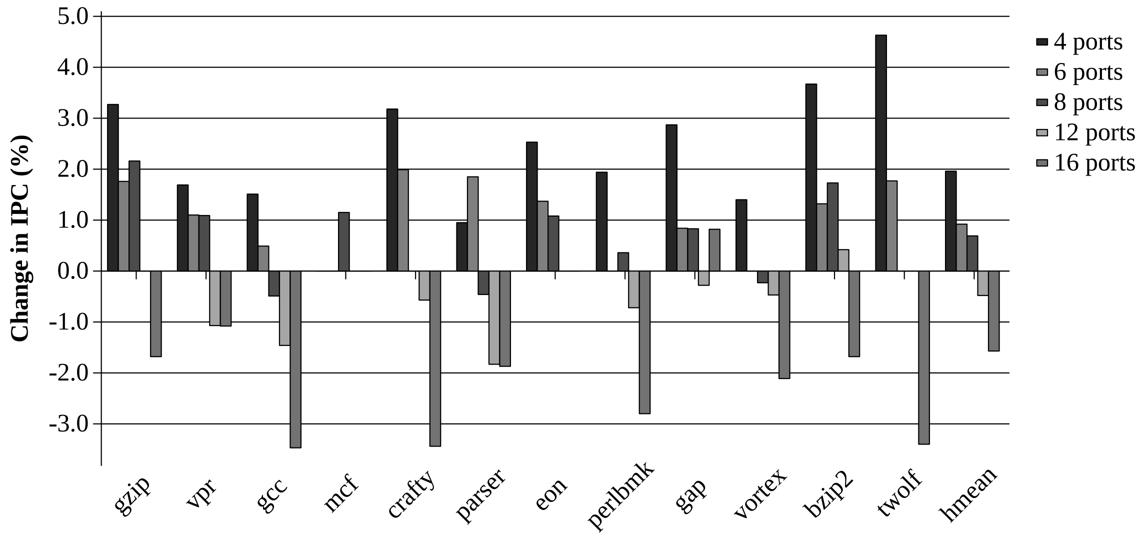


Figure 5.16: Speedups of tree-based steering (TREE) over dependence-based (DEP).

cause they would potentially delay instruction issue. They would also increase the number of steering stages in the pipeline, thus increasing the branch misprediction penalty. During recovery, it is beneficial to issue all instructions in the first fetch group after the mispredicted branch as soon as possible.

5.2.4 Secondary Heuristics

Several additional metrics have been considered in previous steering algorithms such as the DCOUNT steering algorithm. *Source Availability* indicates whether or not an instruction's source operand has already been produced when an instruction is assigned to a cluster. *Cluster Availability* depends on both whether or not a cluster is full, and whether or not there are currently write ports available for a given cluster. This section discusses how these metrics can be used by dependence-based steering algorithms, and how they affect performance.

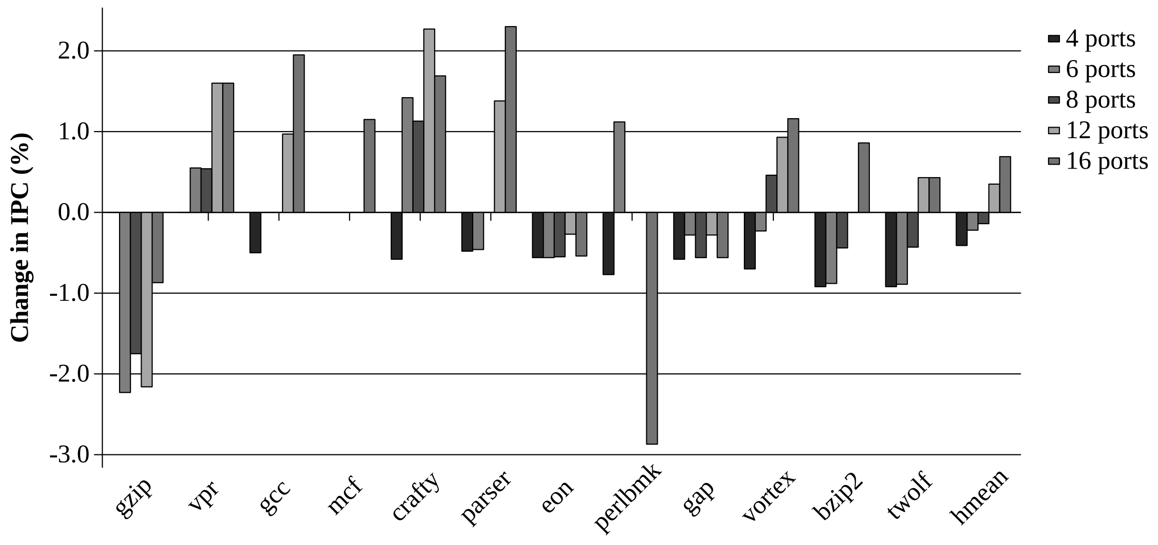


Figure 5.17: Speedups when considering availability of source operands (DEP+`avail` vs. DEP).

5.2.4.1 Source Operand Availability

One important piece of information used in some steering algorithms is whether or not an instruction's source operands are already available at the time an instruction is assigned to a cluster. Many source operands have already been produced when instructions are first placed in the window (see Figure 4.3). If a source operand is already available in all clusters, then there is no need to assign the instruction to the same cluster in which the source operand was produced. Hence, a secondary heuristic, if one exists, can be used for cluster assignment.

Figure 5.17 shows the performance improvement of a dependence-based algorithm that considers source operand availability over a simple dependence-based steering algorithm that does not consider source operand availability. In both algorithms, if an instruction does not have a source operand, the instruction is assigned to a random cluster. Results were shown for five configurations using different numbers of write ports per cluster. Keep in mind that, as shown in Figure 5.7, the configurations with many ports have higher contention for functional units. The source operand availability heuristic is effective at reducing this contention. It allows the steering logic to distribute instructions among the clusters more evenly because the data dependence graph is not strictly used in determining cluster assignments. For instance, the fraction of instructions delayed due to limited functional units is reduced from 8.6% to 6.2% in the 16-port configuration and 7.6% to 5.5% in the 12-port configuration.

5.2.4.2 Cluster Fullness

A second heuristic used by many steering algorithms is Cluster Fullness. This could be simply knowledge of which scheduling window contains the fewest instructions, or a measure of imbalance such as the DCOUNT heuristic. If an instruction has no cluster preference based on source operands, it can be assigned to the least-loaded cluster to improve workload balance. Figure 5.18 shows the performance improvement of a dependence-based steering algorithm that assigns instructions with no source operands to the least-loaded cluster over a dependence-based steering algorithm that assigns instructions with no source operands to a random cluster (same baseline as used in Figure 5.17). This heuristic alone does not add much benefit. There is a small benefit for the configurations with many ports per cluster since these have the most resource contention.

5.2.4.3 Combining the `load` and `avail` Heuristics

The two heuristics previously described, knowledge of the least-loaded cluster combined with knowledge of source operand availability, can be combined for further performance improvements. Figure 5.19 shows the IPC improvements of an algorithm that steers instructions to the cluster where their source operands will be produced, or to the least loaded cluster if either they have no source operands or their source operands have al-

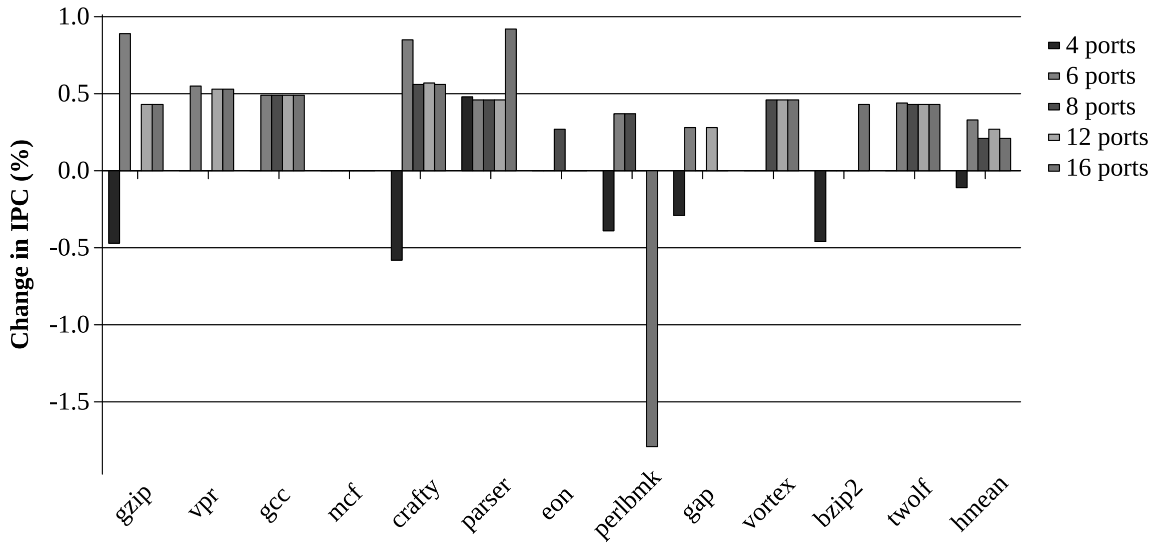


Figure 5.18: Speedups when considering least-loaded cluster (DEP+`load` vs. DEP).

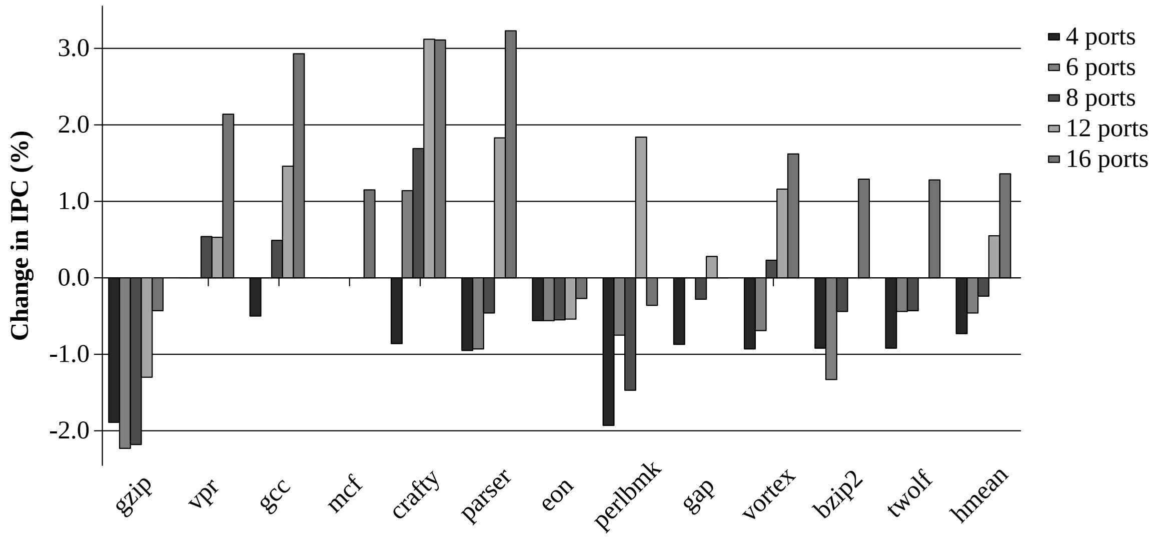


Figure 5.19: Speedups when considering both source availability and the least-loaded cluster (DEP+`load`+`avail` vs. DEP).

ready been produced. This algorithm is compared to the same baseline algorithm as used in Figures 5.17 and 5.18. When these heuristics are combined, they result in super-linear speedups for the configuration with 16 ports per cluster.

5.2.4.4 Last-Source Prediction

With dependence-based algorithms, instructions are steered to the cluster containing their "left" source operand if they have more than one operand, unless an additional heuristic suggests to do otherwise. However, if an instruction is waiting for two source operands to be produced, it is desirable to steer the instruction towards the cluster containing the operand that will be produced last. By predicting which source operand will be produced last and steering instructions toward the cluster containing the predicted-last operand, these instructions can frequently eliminate many inter-cluster forwarding delays.

Simulations were run when adding a last-source predictor [64] to the steering logic. The last-source predictions are made using 2-bit saturating counters stored along with each instruction in the instruction cache. When an instruction is fetched, the upper bit of the counter specifies whether the first or second parent would finish last. The prediction accuracy when using a 4KB Icache is 92%. Figure 5.20 shows the improvement in IPC when adding this predictor (`lsp`) to the DEP algorithm. On average, there was a 1% improvement for the configuration with 4 issue ports per cluster, and a 1.6% improvement for the configuration with 6 issue ports. Despite the modest improvement in IPC, this heuristic did effectively reduce the number of instructions that had to wait because of inter-cluster forwarding delays. This metric is shown in Figure 5.21. The top set of bars in this graph are the fraction of instructions delayed using the original DEP algorithm, and the bottom set of bars are the results when adding last-source prediction (DEP+`lsp`). On average, there was a 14% reduction in the fraction of instructions waiting in the 4-port configuration, and a 21% reduction in the 16-port configuration.

The `lsp` heuristic was also implemented with the DEP+`load`+`avail`, DCOUNT, and TREE algorithms. The performance was increased by 1 to 2% when adding last-

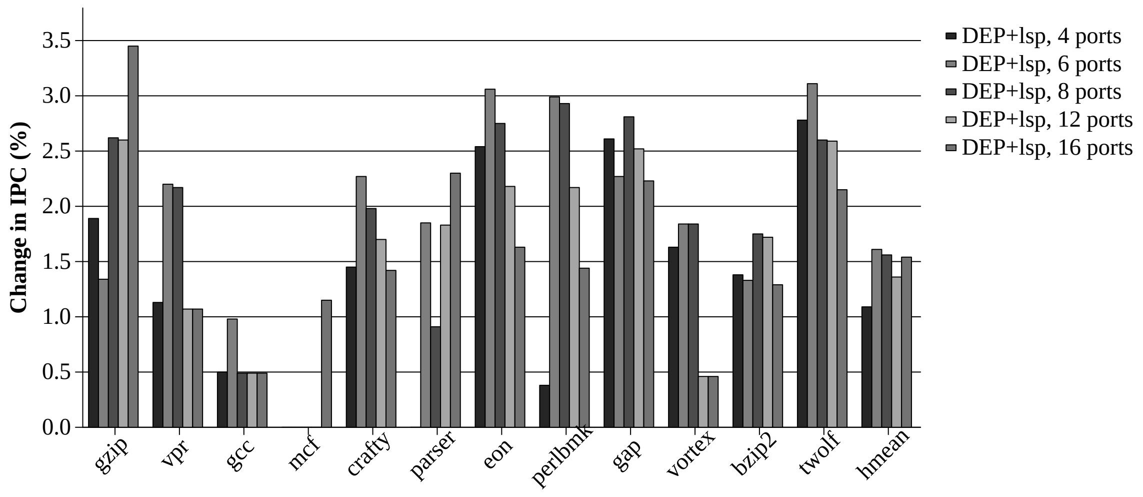


Figure 5.20: Change in IPC when adding last-source prediction to the DEP algorithm (DEP+`lsp` vs. DEP).

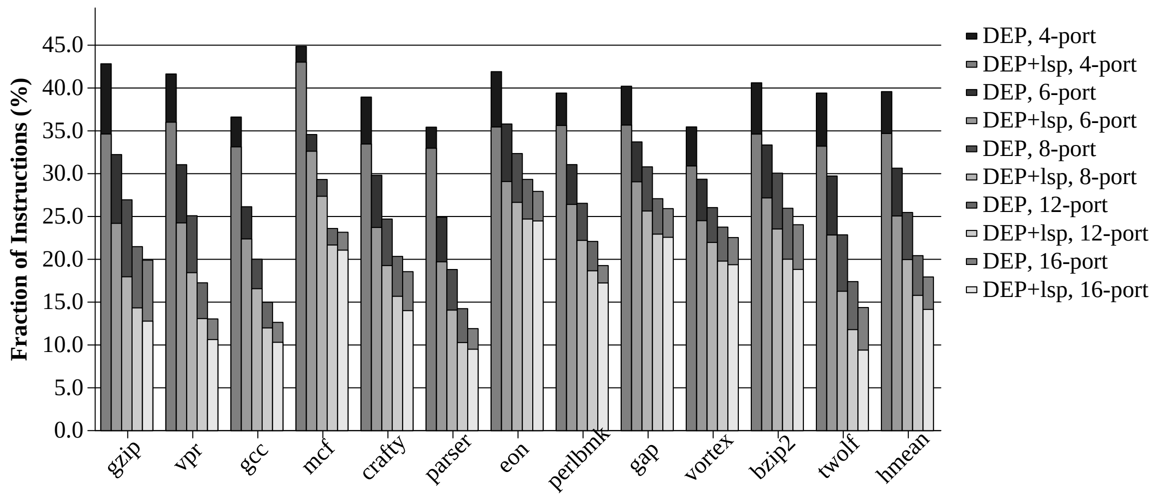


Figure 5.21: Fraction of instructions getting last source from another cluster (DEP, DEP+`lsp`).

source prediction to the DEP+`load`+`avail` algorithm. The performance improvement when adding `lsp` to the TREE algorithm was less than 1%, and there was no noticeable improvement when adding `lsp` to the DCOUNT algorithm because the DCOUNT and

TREE algorithms already incorporate several steering heuristics.

5.2.5 Steer or Stall?

If an instruction cannot be steered to its desired cluster, it could either be steered to an alternate cluster according to a secondary heuristic or the fetch packet could be stalled until the instruction could be steered to its desired cluster. If the instruction is steered to an undesired cluster, it would incur an inter-cluster communication penalty. If the fetch packet were stalled, younger instructions would also be stalled due to the in-order issue constraint. This could also lower window utilization and possibly prevent younger, ready instructions from executing.

The DCOUNT algorithm was investigated using two heuristics for stalling packets: (1) stall a fetch packet when an instruction's desired cluster was full, and (2) stall a fetch packet when there was no available write port for the desired cluster. Both techniques resulted in an IPC degradation. Figure 5.22 shows the change in IPC when adding the first heuristic to the DCOUNT algorithm, compared to the original DCOUNT algorithm in which an instruction is assigned to the least-loaded cluster if its desired cluster is unavailable. This heuristic is effective at reducing the number of inter-cluster communications, as shown in Figure 5.23. The fraction of instructions that are delayed due an inter-cluster bypass is reduced on average by 35% for the 4-port configuration, and 17% for the 16-port configuration. However, window utilization is decreased. Figure 5.24 shows the average number of instructions that are in the instruction window each cycle. (This includes cycles in which the window is empty due to branch mispredictions.) The top bars are the results for the original DCOUNT algorithm, while the bottom bars are the results of the DCOUNT algorithm with the stalling heuristic. The reduction in inter-cluster communication does not make up for this decrease in window utilization.

The second stalling heuristic, stalling when there was no free port for the de-

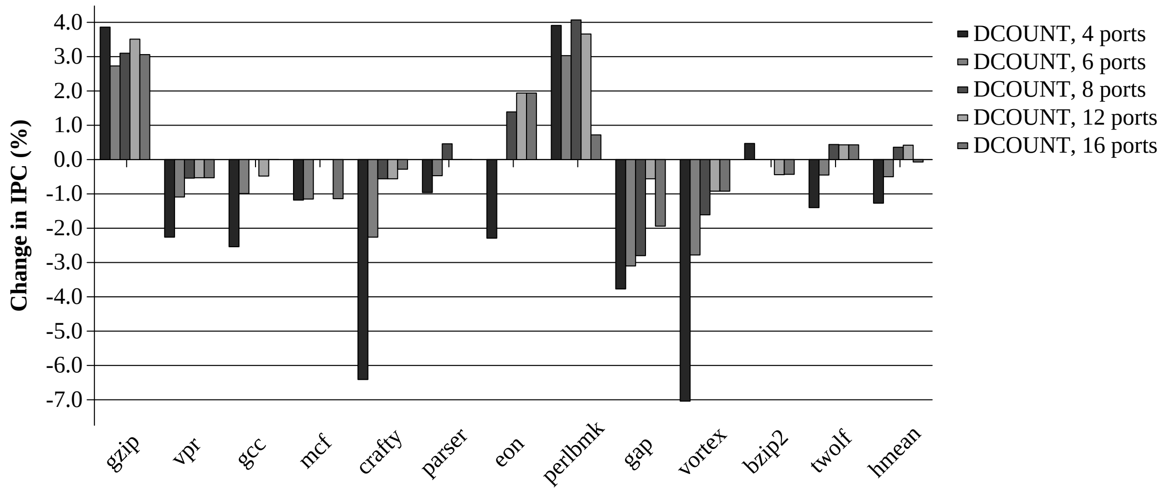


Figure 5.22: Change in IPC when stalling due to a full desired cluster, relative to using DCOUNT secondary heuristics (DCOUNT+S vs. DCOUNT).

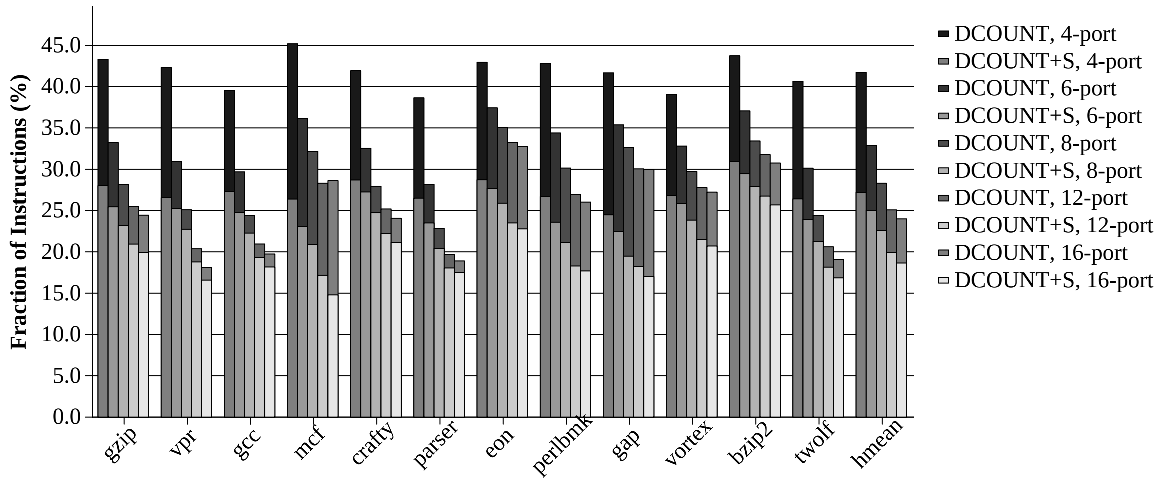


Figure 5.23: Fraction of instructions getting last source from another cluster (DCOUNT).

sired cluster, performed quite poorly with all steering algorithms evaluated. The first stalling heuristic may be useful in clustered paradigms with a severe inter-cluster forwarding penalty, however. This heuristic was evaluated with the PART model presented in Section 4.4.1. This model had a large performance penalty when instructions were

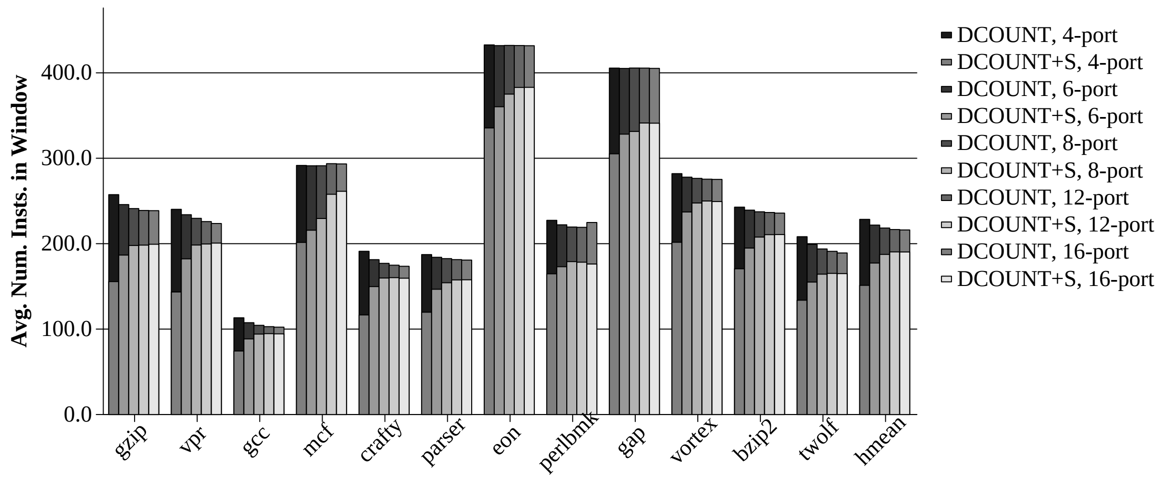


Figure 5.24: Average number of instructions in the window. Top bars: original DCOUNT. Bottom bars: DCOUNT while stalling due to full cluster.

not steered to the same cluster as their source operand. Figure 5.25 shows the results of adding this heuristic to the original dependence-based algorithm with the PART model. The dependence-based algorithm outperformed the DCOUNT algorithm for the PART model, so IPC results for the PART model with DCOUNT steering are not shown.

Note that stalling heuristics should only be used with algorithms that incorporate other heuristics for load balancing. For pure dependence-based algorithms, stalling when the desired cluster is unavailable lowers IPC because there is no mechanism to steer instructions to an alternate cluster, and only one cluster is utilized. Hence, simulation results are only shown for the DCOUNT algorithm.

5.3 Summary and Conclusions

Table 5.3 gives a very brief summary of most of the steering algorithms that have been discussed. The first column gives the name of the algorithm that was used in the graphs. The second and third columns provide a brief summary of what the algorithm does

Name	Description	Strengths	Scale?
MOD	Modulo-N (typically N = num ports)	Load balancing	good
DEP	Dependence-Based	Avoids inter-cluster communications. Poor load balancing	poor
FDRT	Trace-cache algorithm with pinning.	Does not increase pipeline depth.	good
DCOUNT	DEP with load balancing heuristic built in	Balances minimizing inter-cluster communications and load balancing	poor
FDRT+VC	FDRT with virtual cluster renaming	Does not increase pipeline depth. Great with more than minimum ports per cluster	good
2PASS	Out-of-order DEP-based	Avoids inter-cluster communications. Better at load balancing than DEP	poor
TREE	Out-of-order algorithm using dependence trees	Avoids inter-cluster communications	poor
`...+lim-`N	Limit dependence chain depth to N.	Makes a poorly scalable algorithm scalable with performance close to original algorithm.	n/a
`...+avail`	consider source availability Secondary heuristic	Eliminate useless dependence information	n/a
`...+load`	Least-loaded cluster is default. (secondary heuristic)	Improves load balancing	n/a
`...+lsp`	last-source prediction for 2-source instructions	Avoids inter-cluster communications	good
`...+S`	Stall if cluster unavailable. (no secondary heuristic)	Avoids inter-cluster communications	n/a

Table 5.3: Summary of steering algorithms. Top: previously published algorithms. Middle: algorithms introduced in this chapter. Bottom: additional heuristics.

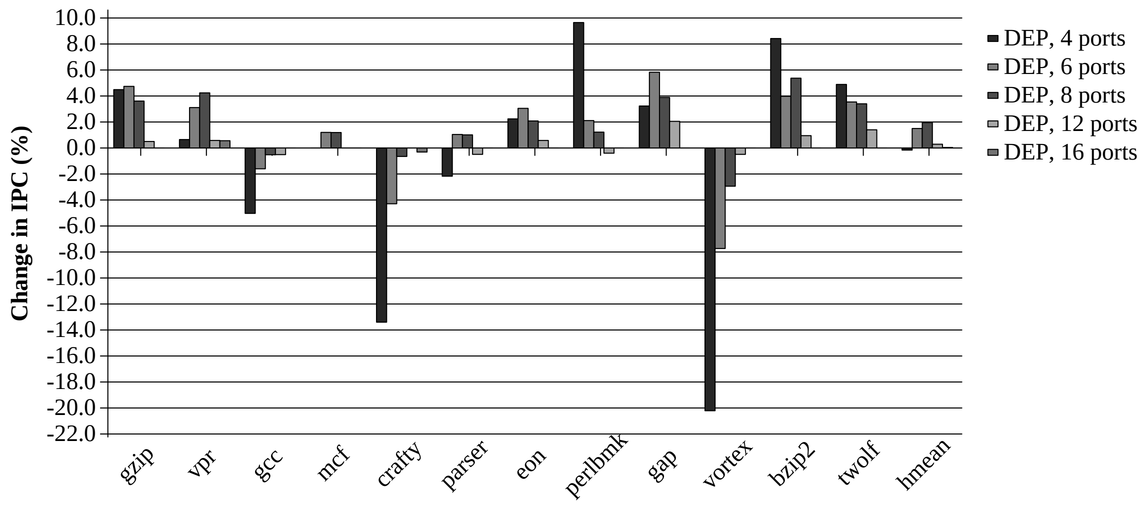


Figure 5.25: PART Model: Change in IPC when stalling due to a full desired cluster (DEP+`S` vs. DEP).

and its strengths. The fourth column in this table indicates if the algorithm is scalable to wide-issue or high-bandwidth machines. The first set of algorithms (MOD, DEP, FDRT, and DCOUNT) are those which were previously published and are discussed in the related work in Section 2.3. Performance results and analysis of these algorithms were presented in Section 5.1. Three additional algorithms (FDRT+VC, 2PASS, and TREE) were introduced in Section 5.2. The FDRT+VC and TREE algorithms use virtual cluster renaming, and the 2PASS and TREE algorithms use out-of-order steering.

The last set of entries in this table are the additional heuristics that can be used in combination with other steering algorithms. The heuristic `lim`-N can be added to any dependence-based algorithm to make it more scalable to wide-issue machines. Scalable implementations of dependence-based algorithms were evaluated in section 5.2.2. The DCOUNT algorithm uses the `avail` and `load` heuristics, but these can be used with other dependence-based algorithms as well. The `lsp` and `S` heuristics can be added to any dependence-based algorithm.

There are three primary benefits that have come from the work presented in this chapter:

- **scalability.** Previous dependence-based algorithms were not scalable to high-bandwidth machines because of the serialization point in the steering logic: an instruction's steering assignment could not be finalized until each previous instruction's steering assignment was finalized. This chapter has introduced two techniques that scale dependence-based algorithms. The first technique is to limit the depth of a chain of dependent instructions that can be steered in one cycle. When limiting the depth to four instructions, there is virtually no impact on IPC regardless of the number of issue ports per cluster. The second technique is to use dependence tree-based algorithms that do not consider inter-fetch-packet dependences. These algorithms are scalable, plus they showed an IPC improvement for machines with few ports per cluster: 2% on average with four ports per cluster.

- **virtual cluster assignments.** Cluster renaming significantly improves the performance of trace cache cluster assignment algorithms by allowing the number of issue ports per cluster to be greater than the minimum while still distributing instructions among all clusters. Improvements of up to 4% are possible when the number of issue ports per cluster is increased to 12. While the windows with 12 issue ports have a latency that is 16% longer than the windows with four ports, the number of entries in the window can be scaled to accommodate the difference in latency. IPC results of scaling the number of entries in a clustered machine were shown in Figure 4.9. Decreasing the number of scheduling window entries from 256 to 192 (which would more than make up for the increase in scheduling latency) would reduce IPC by less than 1%, even with a state-of-the-art hybrid branch predictor.

- **out-of-order steering algorithms.** Out-of-order steering algorithms allow both improvements in load balancing and reduction in inter-cluster communication delays in many configurations. The 2PASS algorithms improved IPC by a modest amount (1.5%) over dependence-based algorithms. The TREE algorithm assigns instructions to clusters out-of-order, and is scalable to high-bandwidth machines.

Clustering paradigms cannot be properly evaluated without using steering algorithms appropriate for each paradigm. Conversely, steering algorithms that perform well with some clustering configurations may perform poorly with others. Consequently, the results shown for each clustering paradigm presented in Chapter 4 were produced using the most appropriate high-performance steering algorithm. All results shown in that chapter use front-end steering algorithms. Front-end algorithms have demonstrated higher performance than history-based algorithms, although using history-based algorithms to store cluster assignments in the trace cache may reduce the latency of the front-end of the pipeline, which in turn can reduce the branch misprediction penalty.

The latency of the scheduling window correlates with the number of issue ports. Using 16 issue ports per cluster has a wakeup and tag broadcast latency that is 24% greater than using four issue ports per cluster. This latency increases two components of the critical scheduling loop: wakeup and tag broadcast. Select latency is unaffected, however. To compensate for the increase in latency, the number of entries in the scheduling window can be reduced to meet cycle time criteria if necessary. The effects on IPC and power of scaling the window sizes were investigated in Chapter 4. For steering algorithms that improve with a high number of ports, it is beneficial to reduce the window size, if necessary, in order to use more issue ports. Power constraints must also be met, however. The effect of scaling the number of ports on power consumption was also investigated in Chapter 4.

Chapter 6

Pipelined Instruction Scheduling

Dynamic instruction scheduling logic allows instructions to execute out of program order as long as all data dependences are obeyed. Palacharla, Jouppi, and Smith [51] first recognized the fact that the scheduling loop may limit performance. The scheduling loop (see Figure 1.5) is a critical loop because an instruction cannot be scheduled until the instructions producing its source operands have been scheduled. Pipelining this loop over multiple cycles prevents data dependent instructions from executing in back-to-back cycles. This is illustrated in Figure 6.1[1]. The SUB, which is dependent on the ADD instruction, cannot be scheduled until after the SUB is scheduled. When the scheduling logic is pipelined over two cycles, IPC will be reduced as shown in Figures 1.2 and 1.3, but the scheduling logic may limit clock frequency if it is not pipelined.

This dissertation proposes *speculative scheduling* as a means of breaking the critical scheduling loop. This chapter evaluates two speculative scheduling techniques that can be used to pipeline the scheduling logic without prohibiting back-to-back data-dependent execution: *speculative wakeup* and *select-free scheduling*. Section 6.1 discusses speculative wakeup, and Section 6.2 discusses select-free scheduling.

[1]In 1-cycle scheduling logic, there is one latch within the wakeup-select-broadcast loop. The location of this latch is dependent on the scheduling logic implementation. Hence, in some pipeline diagrams, wakeup and select will fall in the same cycle, while in others, where a different scheduler implementation is assumed, wakeup and select operations may fall in consecutive cycles. However, the total time for the scheduling loop must still take one cycle in order for dependent instructions to execute in back-to-back cycles.

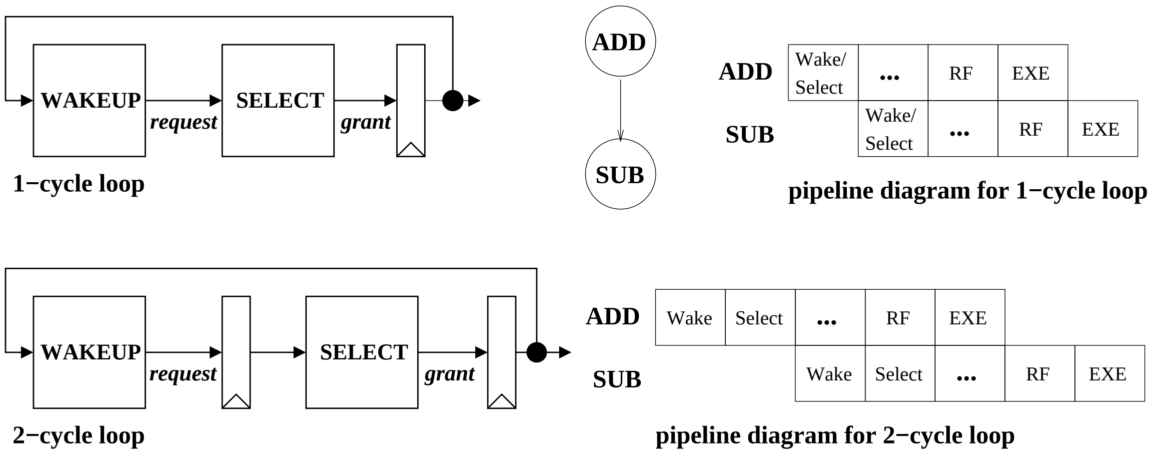


Figure 6.1: One and Two-Cycle Scheduling Loops

6.1 Speculative Wakeup

With conventional dynamic scheduling, an instruction wakes up after its last parent has been selected for execution. With Speculative Wakeup, an instruction assumes that its parents will wake up and be selected for execution immediately after their last parents were selected for execution. For example, in Figure 6.2, the *grandparents* of the SUB instruction are the AND, OR, and XOR instructions. When using Speculative Wakeup, the SUB instruction assumes that the ADD instruction is selected the cycle immediately after the latter of the AND and OR instructions is selected (since the AND and OR are both 1-cycle instructions). It also assumes that the NOT is selected one cycle after the XOR is selected. Speculative Wakeup allows the scheduling logic to be pipelined over two cycles by letting instructions wake up when their last grandparent's tag is broadcast. The scheduling loop is stretched over two cycles while still allowing dependent instructions to execute in back-to-back cycles.

For example, Figure 6.3 shows a pipeline diagram assuming that the AND instruction is the last grandparent of the SUB to broadcast its tag when using conventional 1-cycle scheduling. The arrows represent which tag broadcasts are waking up which instructions.

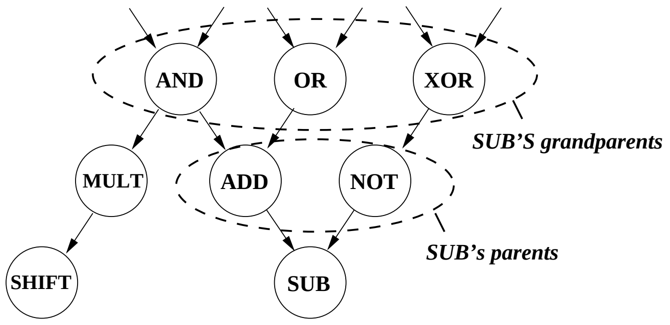
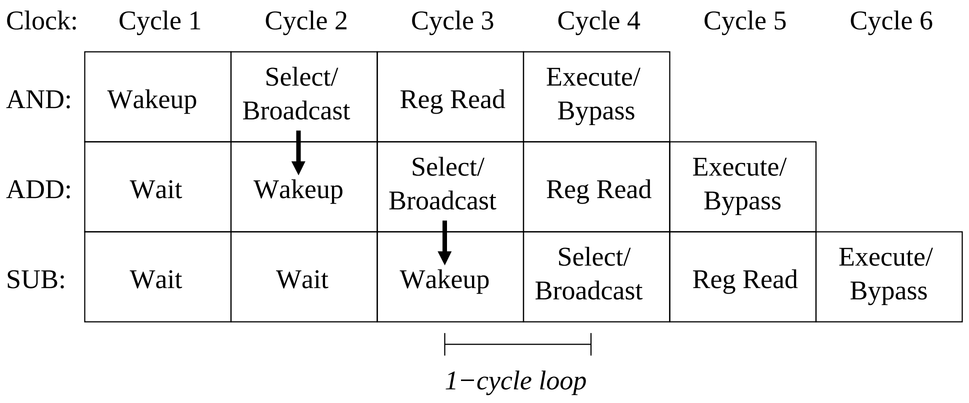


Figure 6.2: A Dependency Graph



Clock:	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6
AND:	Wakeup	Select/ Broadcast	Reg Read	Execute/ Bypass		
ADD:	Wait	Wakeup	Select/ Broadcast	Reg Read	Execute/ Bypass	
SUB:	Wait	Wait	Wakeup	Select/ Broadcast	Reg Read	Execute/ Bypass

1−cycle loop

Figure 6.3: Pipeline Diagram with Conventional Scheduling

Conventional scheduling forms a 1-cycle critical loop because the SUB's tag broadcast is one cycle after its wakeup, and it cannot wake up until the ADD's tag is broadcast. Hence, an instruction's wakeup and select phases must complete within one cycle, even though the clock edge may fall between them.

Figure 6.4 shows the same instructions being scheduled when 2-cycle scheduling with speculative wakeup is used. In cycle 3, the SUB instruction wakes up when the AND instruction broadcasts its tag. In this case, the scheduling logic forms a 2-cycle loop because the SUB's tag is broadcast two cycles after it wakes up. However, a chain of dependent instructions are still scheduled in back-to-back cycles.

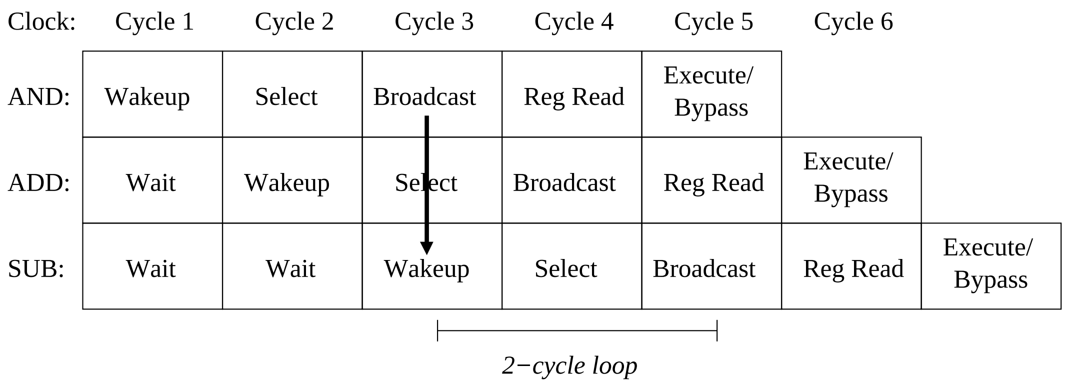


Figure 6.4: Pipeline Diagram with Speculative Wakeup

As long as an instruction's parents are single-cycle instructions, it will wake up when its last grandparent tag is broadcast when using 2-cycle pipelined scheduling. However, if a parent is a multi-cycle instruction, the parent's source tags are not needed by the child. If the parent takes N cycles to execute, the child will wake up N-2 cycles after the parent's tag is broadcast (whereas with traditional 1-cycle scheduling the child would wake up N-1 cycles after its parent's tag was broadcast). For example, if the MULT instruction in Figure 6.2 has a 4-cycle execution latency, the SHIFT instruction would wake up two cycles after the MULT's tag broadcast.

6.1.1 False Selection

An instruction may wake up based on its grandparents' tags even if one of its parents is delayed due to functional unit contention. A *false selection* would occur if that instruction were selected for execution. Suppose the ADD was not selected for execution in Cycle 3 of Figure 6.4. The SUB would still wake up and request execution since the AND's tag was broadcast in cycle 3. A false selection would occur if the SUB were selected for execution. False selections only affect performance if the instructions that are falsely selected prevent really ready instructions from being selected for execution. The mechanism to detect false selections is explained in the following section.

6.1.2 Implementation

To support speculative wakeup, the rename logic must be modified to track grandparent dependences, and the wakeup logic must be modified to hold additional operand tags. This section explains the modifications.

6.1.2.1 Rename Logic

For pipelined scheduling with speculative wakeup, the rename logic is responsible for determining the destination tags of an instruction's grandparents as well as the destination tags of its parents. Note that the destination tags of an instruction's grandparents are the same as the source operand tags of the parents. In order to determine the source operand tags of the parents, the Register Alias Table must be augmented. Each map entry is extended so that, in addition to the original physical register number, it contains the set of physical register numbers needed to produce the original physical register number. In other words, an entry contains the destination tag and source operand tags of the instruction that updated the entry.

In addition to augmenting the Register Alias Table, the dependency analysis logic must be modified. An instruction must obtain its parents' source tags as well as the parents' destination tags. Two multiplexors, called *grandparent multiplexors* are required for each instruction being renamed (except for the first two instructions in the fetch group). One example of such a multiplexor is shown in Figure 6.5. This is one of two grandparent multiplexors for `Op3`, the third instruction in the fetch group. The first input to the MUX, labeled `Parent_Preg(Op3_src1)`, is the pair of source operand tags from the RAT entry of `Op3`'s first architectural source register. These two tags are selected when the grandparent was in a previous fetch group (i.e. not instructions `Op1` or `Op2`). The next two pair of tags are the source operand tags for the first two instructions in the fetch group. These tags are the outputs of the dependency analysis logic shown in Figure 3.3. The control for this MUX is the same as the control used for the MUX to select the third instruction's first source tag shown in Figure 3.3. The grandparent multiplexors have the same number of inputs as the source operand multiplexors, but each input contains two tags instead of one. These muxes add at most one multiplexor input-to-output delay to the critical path of the rename logic.

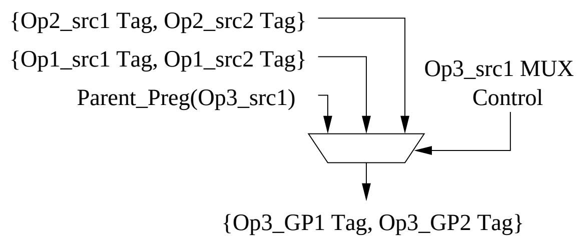

{Op2_src1 Tag, Op2_src2 Tag}
{Op1_src1 Tag, Op1_src2 Tag}
Parent_Preg(Op3_src1)
Op3_src1 MUX Control
{Op3_GP1 Tag, Op3_GP2 Tag}

Figure 6.5: A Grandparent MUX for the 3rd Instruction in a Packet

6.1.2.2 Wakeup Logic

Even though instructions wake up using their grandparent tags when their parents are single-cycle instructions, the parent tags are still needed to detect false wakeups. Hence, the CAM arrays must store up to six tags per instruction (assuming two parents per instruction). This may increase the area of the scheduling window unless the window is wire-bound from all of the tag buses. Section 6.2 discusses a simplification to save area where fewer tags are needed.

The modified scheduling logic is shown in Figure 6.6. For simplicity, only the Ready bits are shown. Other fields for each tag are similar to the ones in the original scheduling logic shown in Figure 3.4. If a parent is a single-cycle instruction, then the corresponding grandparent pair of tags can be marked invalid (Ready bit set). An instruction

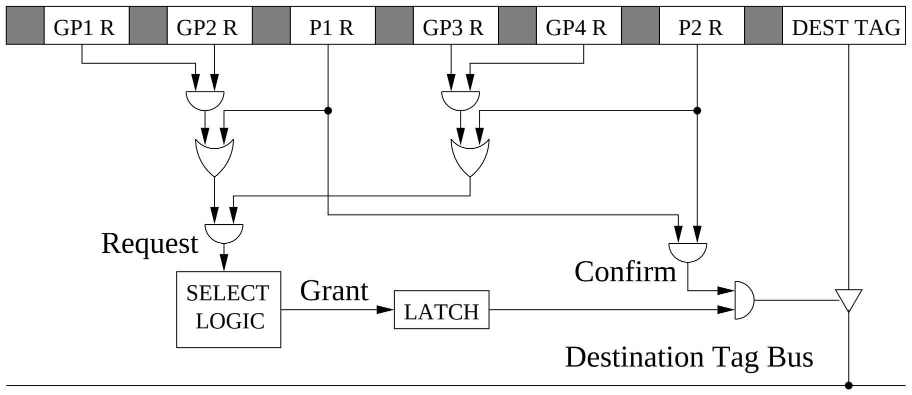


Figure 6.6: Speculative Scheduling Logic

requests execution when, for each parent, either the parent is ready or the parent's parents are ready.

The Confirm line is used to verify that an instruction is really ready to execute and is not a false selection. The confirm line is only asserted when all of the instruction's parents (not just grandparents) are ready. Assuming the parents are single-cycle instructions and are not stalled, it is asserted one cycle after the request line. If the instruction is granted execution and the Confirm line is asserted, then the instruction's destination tag is broadcast. An instruction cannot be removed from the window until after its grant is confirmed. This may keep instructions in the scheduling window one cycle longer than with a conventional scheduler.

6.1.3 Preventing Deadlock

Suppose the ADD was not actually selected for execution in Cycle 3 of Figure 6.4. Then in Cycle 4, both the ADD and its child, the SUB, would request execution. If the SUB had higher priority than the ADD, then deadlock may occur. In order to prevent deadlock, the select logic must use a priority such as oldest-first, or a rotating priority so that the ADD will not be starved of execution. With oldest-first selection priority, a child will not be falsely selected for execution before the parent is selected, although they may both be selected in the same cycle. If they were both selected in the same cycle, the child's execution grant would not be confirmed because it had not yet received its parent's scheduling tag, so the child would remain in the scheduling window and would not broadcast its tag. When the scheduling window is clustered and the parent and child reside in different clusters, the child may be granted execution before its parent. However, as before, the child will remain in the window because its execution grant was not confirmed. As long as all clusters schedule instructions in program order, the parent will eventually be selected for execution. Oldest-first priority is assumed in all simulations, although a round-robin priority mechanism would work as well.

6.1.4 Last-Source Prediction

Frequently, for instructions with two parents, it is easy to predict which source will be broadcast last. Hence, it is easy to predict which pair of grandparents will broadcast the last grandparent tag. With Last-Source Prediction, only the parents of the predicted-last parent are stored in the scheduling entry. Hence, there are four tags per entry rather than six: both parents, as needed for confirmation, and the two parents of the parent predicted to be last. This will reduce the area of the scheduling window if it is in fact wire-bound.

The last-source predictions are made using 2-bit saturating counters stored along with each instruction in the instruction cache. When an instruction is fetched, the upper bit of the counter specifies whether the first or second parent would finish last. If the predictions are incorrect, instructions will eventually wake up when both parents have broadcast their tags. At worst, this delays an instruction's execution by one cycle. The prediction accuracy when using a 4KB Icache is 92%, which is good enough given a 1-cycle misprediction penalty. Performance results using last-source prediction are shown in Section 6.1.5.

6.1.5 Performance Results

Speculative wakeup was evaluated using four machine configurations: (1) an unclustered 4-wide pipeline, (2) an unclustered 8-wide pipeline, (3) an unclustered 16-wide pipeline, and (4) a 16-wide pipeline with four clusters using the DCOUNT steering algorithm. The configurations with narrow execution widths are provided just to see the effects of functional unit contention on scheduling behavior. All configurations have the same size scheduling and instruction windows, so the narrow-width machines have a large amount of functional unit contention. Figure 6.7 shows the difference in IPC of Speculative wakeup versus conventional scheduling for these four configurations. The machines with speculative wakeup have the same branch misprediction penalties as the machines with conventional scheduling so that the results are not primarily determined by branch prediction accuracy. For many benchmarks, there was little to no change in IPC so the bars are not visible. The 4-wide configuration had about a 1% IPC degradation, while the others had almost no degradation.[2]

False selections only occur when an instruction's execution is delayed due to functional unit contention. Figure 6.8 shows the fraction of retired instructions that were stalled one or more cycles. The 4-wide configuration had the most contention for functional units. The 16-wide clustered configuration had more stalls than the 16-wide unclustered configuration because instructions could only be scheduled on one of the four functional units

[2]The anomaly in perl is due to the fact that the indirect branch target address misprediction rate decreases relatively by 15% (from 16.6% to 14.1%) when using a pipelined scheduler. This decrease in target address misprediction rate in turn decreases the total number of cycles spent on the wrong path by 8%. To confirm this, simulations were run using perfect indirect branch target address prediction. When this was used, speculative wakeup performed 0.5% lower than conventional scheduling. The reason the target prediction rate can change when changing the scheduling latency is that the indirect branch target buffer gets updated as soon as the target address is known, before instructions retire. This means that wrong-path branches may possibly update this buffer, and correct-path instructions may update this buffer out-of-order since they execute out-of-order. This anomaly shows up again in Figure 6.11.

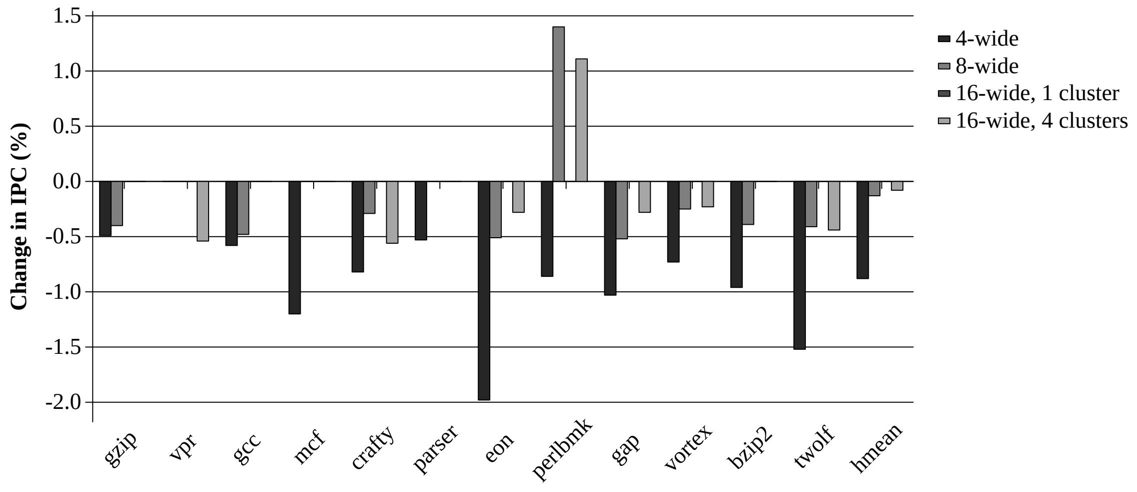


Figure 6.7: Performance of Speculative Wakeup compared to Conventional Scheduling.

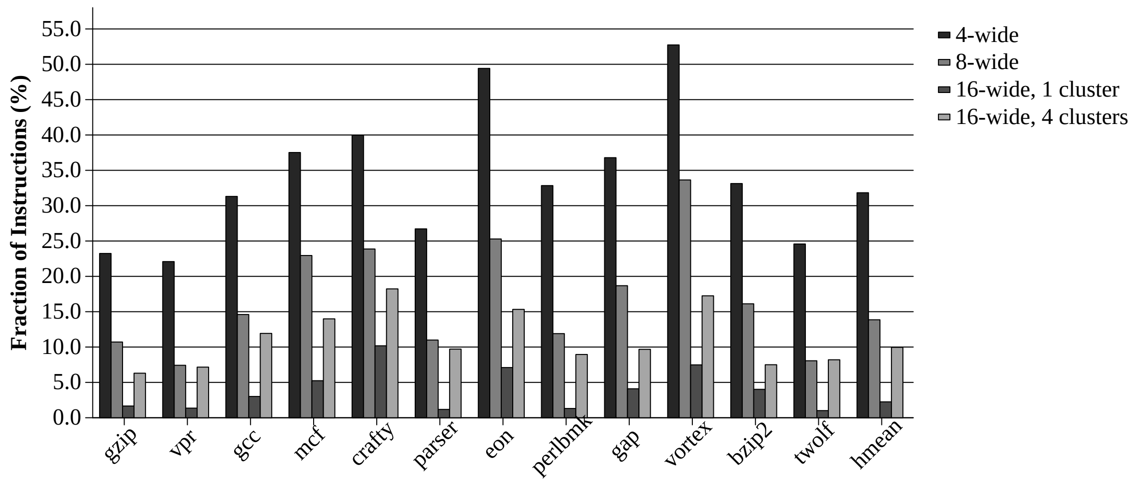


Figure 6.8: Fraction of Retired Instructions with delayed execution due to functional unit contention.

within their cluster. The higher the contention for functional units, the greater the chance that instructions will wake up before they are really ready to execute.

Not all false selections prevent really ready instructions from being selected for execution. If an instruction was falsely selected for execution but it did not contend with a really ready instruction for a functional unit, then it has no impact on performance. Hence, the 16-wide configuration is less likely to have instructions being stalled due to other instructions being falsely selected. Figure 6.9 shows two overlapped bars for each simulation: the top bar is the number of times, per instruction, that an instruction on the correct path was falsely selected. The bottom bar is the number of times, per instruction, that a false selection prevented a really ready correct-path instruction from executing. Of the non-clustered machines, the 4-wide configuration had the most false selections because it had the highest contention for functional units. However, even though the 4-wide configuration had over 2.3 times as much contention as the 8-wide configuration, it only had 36% more false selections. This is because the scheduler selects instructions in program order, which means many of the instructions that woke up too early in the 4-wide machine were not selected for execution anyway.

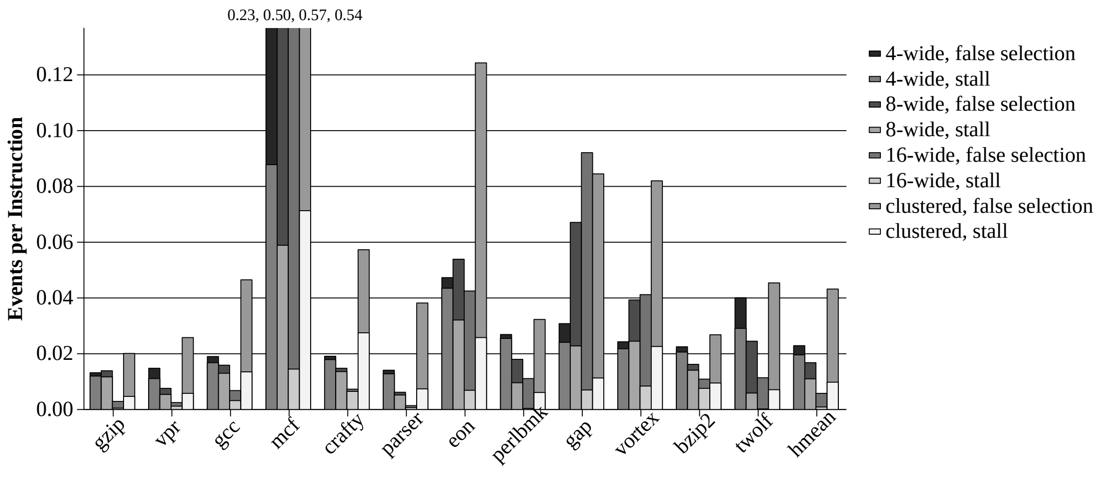


Figure 6.9: False Selections (top) and Scheduling Stalls (bottom).

In the 4-wide configuration, most of the false selections resulted in another really ready instruction being stalled because the execution units were a tight bottleneck. In the 16-wide configuration, fewer of the false selections resulted in execution stalls of really ready instructions. In the benchmarks gap and vortex, 16-wide configuration had more false selections because there were enough functional units to select the younger instructions that woke up too early for execution. Because the clustered configuration allowed children to be falsely selected for execution before their parents, it had far more false selections than the other configurations with strictly in-order scheduling.

The mcf benchmark had an unusually high number of false selections because of instruction replays. The replays were caused by loads that were delayed because of contention for the data cache read ports. Dependants of the load instructions were scheduled before the loads were known to have been delayed. Most of the false selections in mcf did not affect performance, however. In fact, generally most of the false selections are caused by few instructions that are falsely selected over and over. Figure 6.10 shows the percentage of retired instructions that were falsely selected one or more times. When compared to Figure 6.9, one can conclude that some instructions are falsely selected many times.

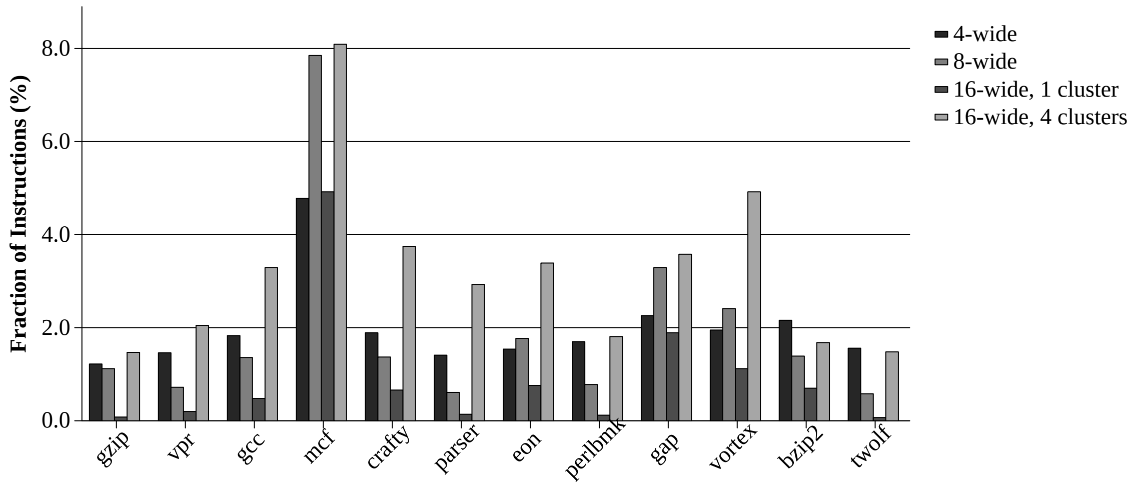


Figure 6.10: Fraction of Retired Instructions that were Falsely Selected.

6.1.5.1 Performance with Last-Source Prediction

Figure 6.11 shows the IPC degradation of speculative scheduling with a limit of two grandparents as compared to conventional scheduling. There was almost no change because most instructions don't need to use four grandparent operands. Figure 6.12 shows the fraction of instructions that had to actually reduce the number of grandparents used for scheduling. Only about 18% of instructions had two single-cycle parents and at least three grandparents that had not yet broadcast their tags to the BBT in the issue stage. Even for the instructions that would have used three or more grandparents when not using last-source prediction, the prediction accuracy of the last-source predictor, as shown in Figure 6.13, was adequate enough to prevent IPC degradation. Figure 6.14 shows the number of false selections per instruction. This is considerably higher than when using regular speculative wakeup because if the last-source prediction is wrong, an instruction may wake up several cycles earlier than it should have. As with speculative wakeup without last-source prediction, most instructions that are falsely selected are falsely selected several times, as one can see by comparing the number of false selections per instruction to the number of instructions that were falsely selected in Figure 6.15.

When using speculative wakeup with last-source prediction, each entry of the scheduling window must have CAMs for four tags instead of just the two which are needed in the baseline scheduler. The impact of the additional CAMs and tags on the latency of the scheduling window is highly dependent on the scheduling window configuration. For a unified 256-entry window with 16 tag broadcasts and 16 write ports, the wakeup latency is increased by 85%. When the window is partitioned over 4 clusters and each cluster has only 4 issue ports, the increase in latency is only 4.7% because the area is bound by the wires of the tag buses, not the CAM logic. Regardless of the configuration, because the wakeup and tag broadcast comprise only a portion of the clock cycle (around a half for many configurations [51]), additional time is available when pipelining the scheduling logic over two

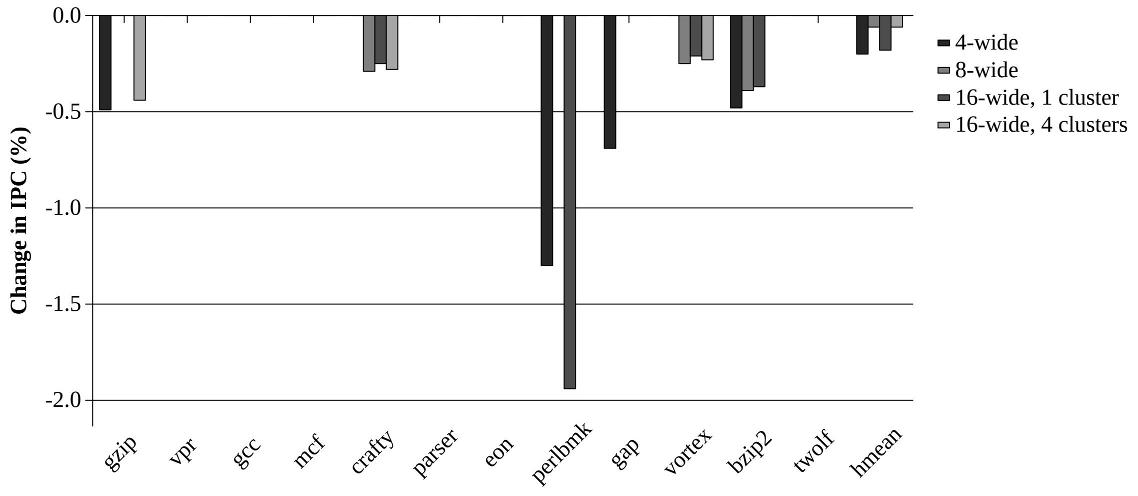


Figure 6.11: Relative IPC when using two grandparents compared to four grandparents.

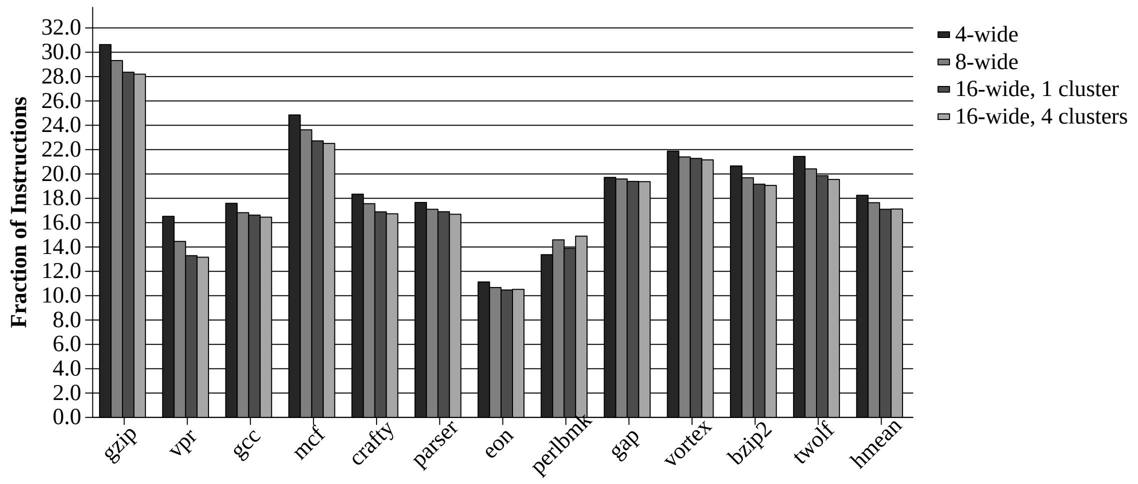


Figure 6.12: Fraction of instructions limiting the number of grandparent scheduling tags.

cycles. This slack could be used to increase the size of the scheduling window to get gains in IPC that will make up for the degradation in IPC caused by false selections. Further reduction in the number of tags is possible as well [21].

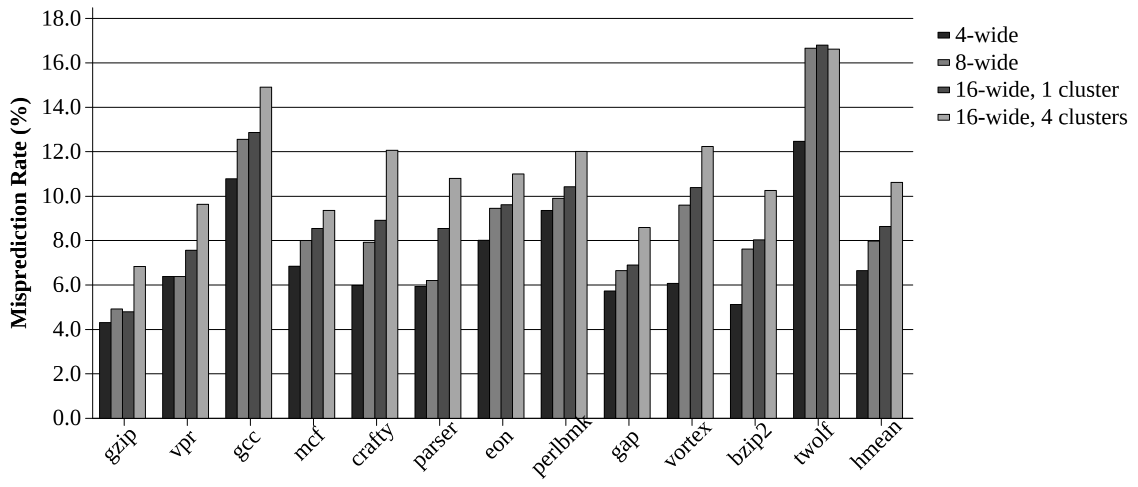


Figure 6.13: Misprediction rate of the Last-Source Predictor.

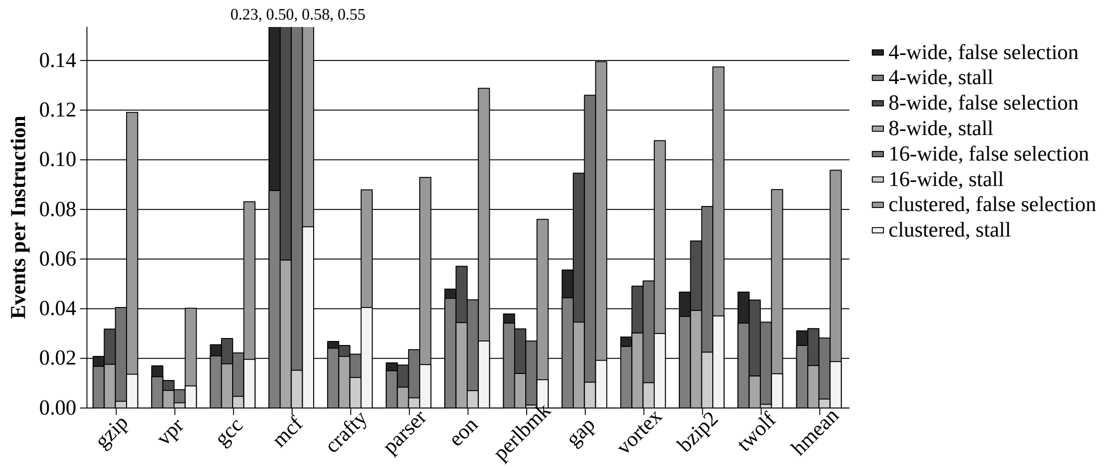


Figure 6.14: False Selections (top) and Scheduling Stalls (bottom), 2 grandparents.

6.2 Select-Free Scheduling

Select-free scheduling exploits the fact that most instructions are selected for execution the first cycle that they request execution. With select-free scheduling, instructions are predicted to be selected for execution the first cycle they are awake, so it is a form of speculative scheduling. Instructions broadcast their tags as soon as they wake up and

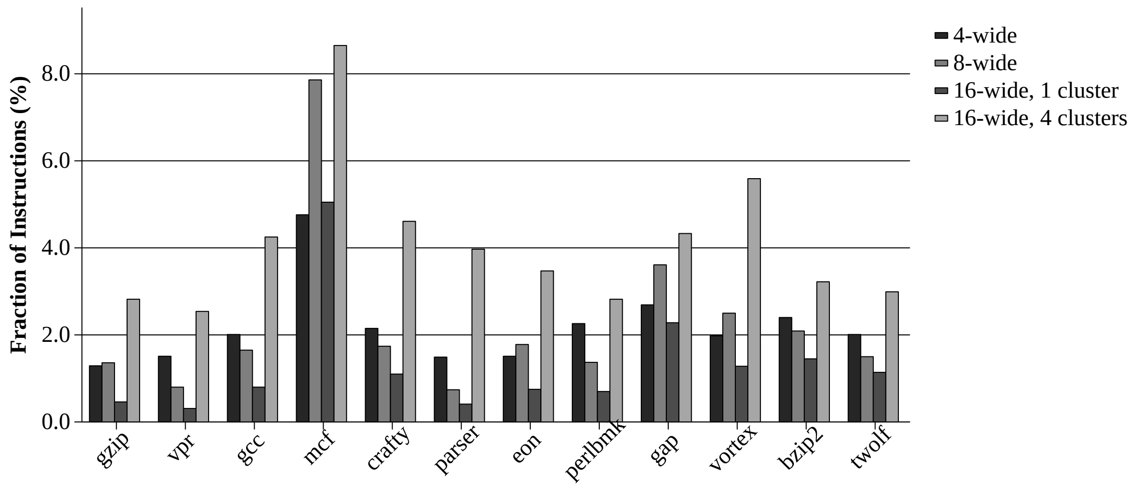


Figure 6.15: Fraction of Retired Instructions that were Falsely Selected, 2 grandparents.

request execution, before their selection has been confirmed. This reduces the amount of logic in the critical scheduling loop. The *select-free* loop contains only the wakeup logic and tag broadcast. This shortened scheduling loop is shown in Figure 6.16. The selection logic may be pipelined over multiple cycles if necessary without affecting the execution of dependent instructions in back-to-back cycles.

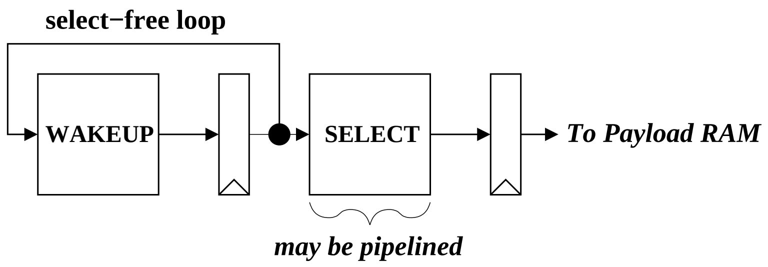


Figure 6.16: Select-Free Scheduling Loop

6.2.1 Collisions and Pileups

With select-free scheduling, instructions may broadcast their tags even when they are not granted execution. A *collision* occurs when more instructions wake up than can be selected, resulting in a misspeculation. For example, in Figure 6.17, suppose the MULT and the SHIFT both wake up in the same cycle after the SUB's tag is broadcast, but only the MULT is selected for execution. The SHIFT is a collision victim and broadcasts its tag, speculating that it is selected for execution, even though it is not.

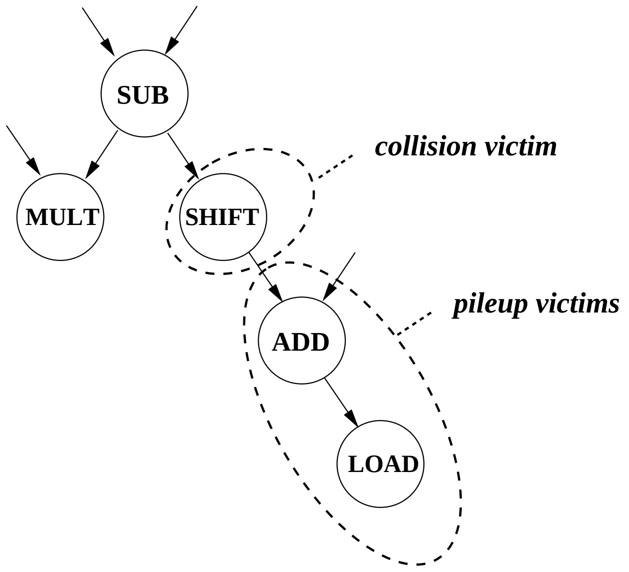


Figure 6.17: Dependency Graph with Collision and Pileup Victims

Because a collision victim broadcasts its tag too early, its dependants wake up and request execution too early. These instructions are called *pileup victims*. For example, the ADD instruction will wake up when the SHIFT broadcasts its tag and requests execution before it is really ready. The ADD in turn broadcasts its tag too early, causing the LOAD to also wake up too early. Hence, a collision can cause a chain of pileups. The next section explains how collision and pileup victims are detected and rescheduled.

6.2.2 Implementation

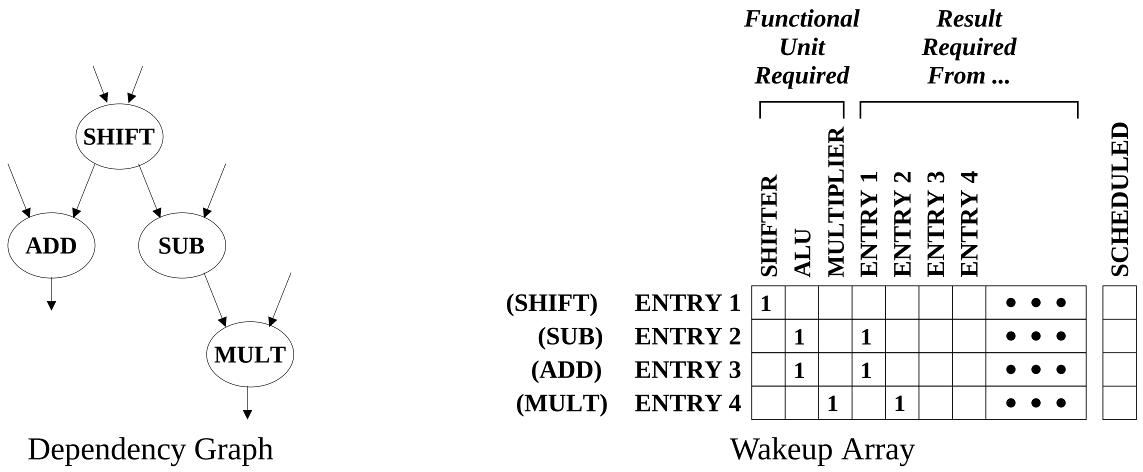
6.2.3 Array Schedulers

Select-free scheduling is implemented using an array-style scheduling window. Dependences are indicated using a dependency matrix rather than source operand tags. Dependency matrices were originally used in the context of scheduling to schedule memory operations [56]. Recently, they have been used for scheduling register-register instructions as well, and have been shown to have lower latency than CAM implementations for some configurations [32].

With a dependency matrix, there is one row for each instruction waiting in the window. This row contains a bit vector that indicates which resources are necessary for the instruction to execute. Each bit corresponds to a particular resource. Resources may be either source operands or hardware resources, such as functional units.

For example, Figure 6.18 shows a dependency graph and the contents of a wakeup array for the instructions in the dependency graph. The instructions in entries 1–4 are the SHIFT, SUB, ADD, and MULT instructions from the dependency graph. In this example, the instructions that produced the values for the unspecified source operands of the SHIFT, SUB, ADD, and MULT are available from the register file. The SHIFT instruction only requires the shifter, so only the bit corresponding to the shifter is set. The SUB and ADD instructions depend on the result of the SHIFT and require the ALU, and the MULT instruction depends on the result of the SUB and requires the multiplier. Instructions request execution when all required resources are available and the SCHEDULED bit is not set.

Running vertically across the array are wires, called AVAILABLE lines, which indicate which resources are available. The scheduling logic for one entry is shown in Figure 6.19. A wired-OR circuit may be used for this implementation. When the AVAILABLE lines of all required resources (as indicated by set bits of the row) have been asserted and the SCHEDULED bit is not set, the instruction requests execution. The select logic is a



		SHIFTER	ALU	MULTIPLIER	ENTRY 1	ENTRY 2	ENTRY 3	ENTRY 4		SCHEDULED
(SHIFT)	ENTRY 1	1							• • •	
(SUB)	ENTRY 2		1		1				• • •	
(ADD)	ENTRY 3		1		1				• • •	
(MULT)	ENTRY 4			1		1			• • •	

Figure 6.18: Dependency Graph and Corresponding Wakeup Array Contents.

priority circuit as in the CAM-based scheduler implementation. The output of the select logic is used to set the SCHEDULED bit so that the instruction will not repeatedly request execution until it is removed from the window. It also triggers the countdown timer associated with that entry to count the latency of the instruction's execution, similar in behavior to the COUNTER field of the CAM-based scheduler. After $N - 1$ cycles (assuming an N-cycle latency) the AVAILABLE line associated with that instruction is set. The output of the select logic also feeds the Payload RAM. In the previous example, there was a 1-to-1 correspondence between the rows of the scheduler and the AVAILABLE lines of source operands, although this does not have to be the case.

When an instruction's wakeup array entry is deallocated, it may still have dependent instructions residing in the wakeup array. Because entries are deallocated soon after they are scheduled, it is possible that the entry of one of its source operands may be re-allocated to a younger instruction. Several implementations exist that avoid incorrect dependences. For example, when an entry is deallocated, every wakeup array entry in the scheduling window clears the bit corresponding to the deallocated entry.

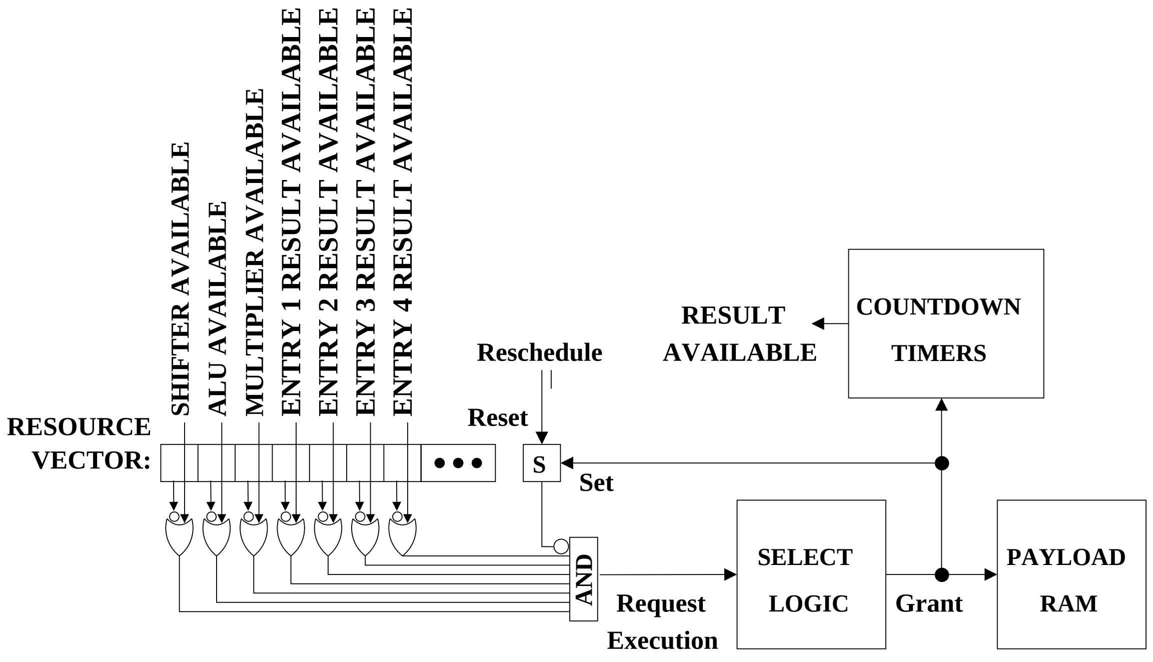


Figure 6.19: Logic for One Wakeup Array Entry

6.2.4 Select-Free Implementation

With a matrix scheduler, there is no limit on the number of instructions that can broadcast their “tags” (which are actually AVAILABLE lines) each cycle. To implement a select-free scheduling loop using a matrix scheduler, the pipeline must be modified to detect and reschedule collision and pileup victims. Figure 6.20 shows the modified pipeline. Two new structures are added. The Collision Detection Logic identifies the collision victims. This circuit is similar to the select logic, which is a priority circuit, except that rather than selecting the N (where N is the number of instructions that can be selected) highest priority requesting instructions, all requesting instructions *except* for the N highest priority instructions are selected. These instructions must be rescheduled, so the SCHEDULED bits of their scheduling entries (labeled “S” in Figure 6.19) are reset.

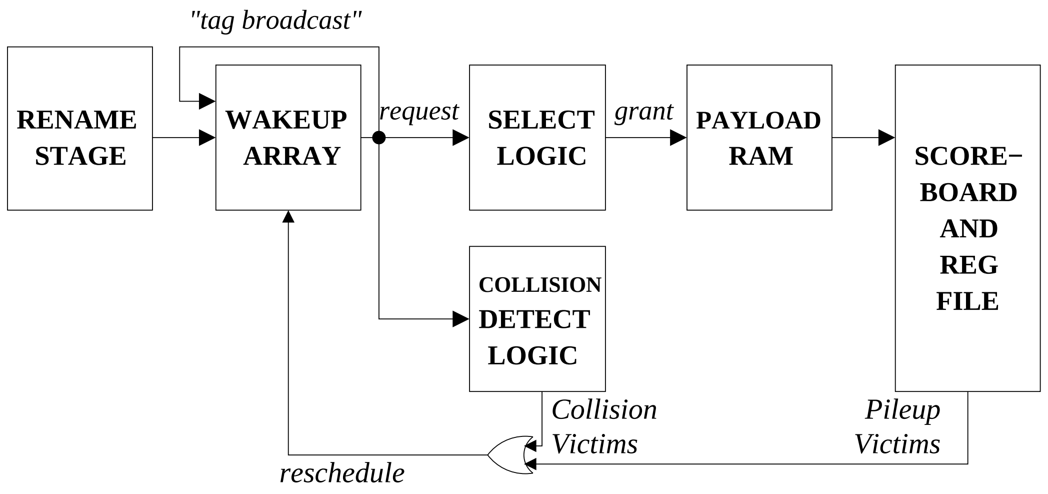


Figure 6.20: Schedulers with support for Select-Free Scheduling

The second structure added to the pipeline is a scoreboard. The scoreboard is used to detect pileup victims. The scoreboard contains one bit per physical register that indicates if the instruction producing that physical register has been successfully scheduled. After an instruction is selected for execution, it accesses the Payload RAM to get the physical register identifiers of its source operands. It uses those identifiers to index the scoreboard. If the scoreboard bits of all source operands have been set, then the instruction is not a pileup victim, and it sets the bit corresponding to its own destination register. Because the scoreboard is accessed in parallel with the physical register file, it does not add any latency to the pipeline.

After an instruction has been identified as a collision or pileup victim, it must be rescheduled. Note that because modern schedulers speculate that loads will hit in the Level-1 data cache and schedule dependent instructions accordingly, they must already have support for rescheduling instructions [41, 45, 69]. The delay for detecting collision victims is similar to the latency of the select logic. The pileup victims cannot be detected until after the Payload RAM has been accessed; hence, it takes longer to reschedule them.

With the replay implementation used in all models, instructions are not removed from the scheduling window until their execution has been confirmed. Hence, select-free scheduling adds some pressure on the scheduling window since instructions that misspeculate must reside in the window for a longer period of time. This may not be necessary in alternative implementations of replay where instructions are removed and placed into a separate buffer as soon as they are scheduled, and re-inserted upon misspeculation [45].

6.2.5 Performance Results

Figure 6.21 shows the impact on IPC when using select-free schedulers for 4-wide, 8-wide, and 16-wide machines as well as a 16-wide machine with four clusters. [3] These are the same configurations as used in Section 6.1.5. The 4-wide machine had a 3% impact on IPC, while the 8-wide and clustered machines both had about a 1% impact on IPC on average. The reason that the 4-wide machine had the biggest performance impact is that it had the most collision victims, as shown in Figure 6.22. Figure 6.23 shows the fraction of instructions that are pileup victims. A collision victim may cause zero, one, or multiple instructions to become pileup victims. On average, there were 0.65 pileup victims per collision victim. The 16-wide machine had the most pileup victims per collision victim (0.79) while the 4-wide machine had the fewest (0.46). This is because contention for functional units was high enough in the 4-wide configuration that many of the “would-be” pileup-victims were not selected for execution. By the time they were actually selected for execution, their parents had actually been selected for execution.

[3] In the mcf benchmark when using the 8-wide configuration, the baseline machine actually has more contention for data cache ports.

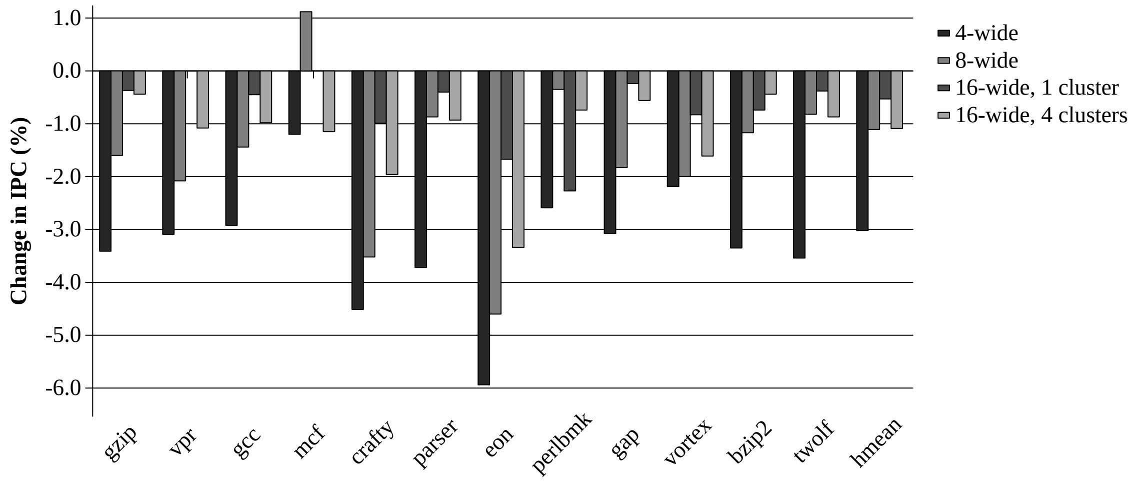


Figure 6.21: Performance of Select-Free Scheduling compared to Conventional Scheduling.

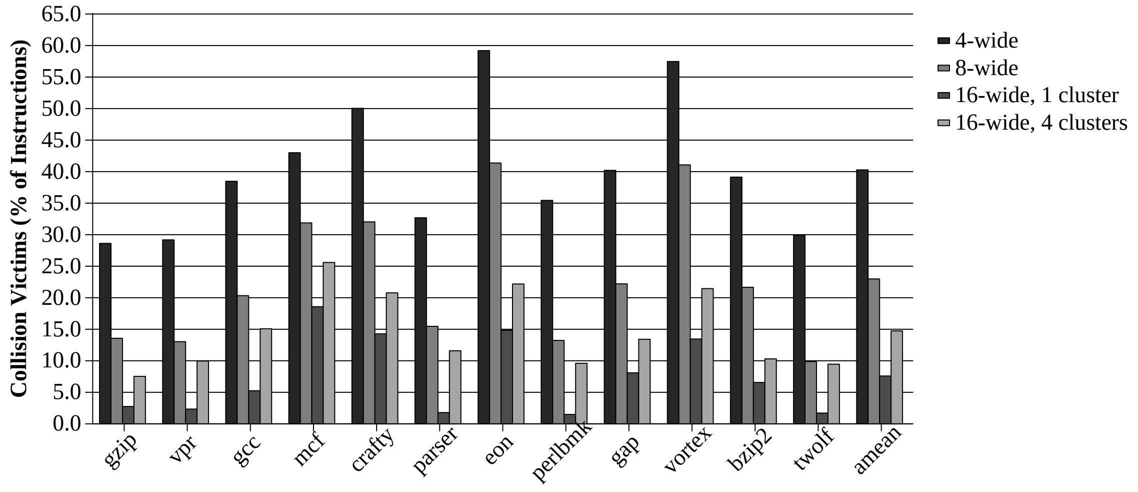


Figure 6.22: Retired Collision Victims.

6.3 Conclusions

Both Speculative Wakeup and Select-Free Scheduling are effective techniques for breaking the critical scheduling loop. Without these techniques, the degradation in IPC when pipelining the scheduling loop is 10% as was shown in Chapter 1. However, both of

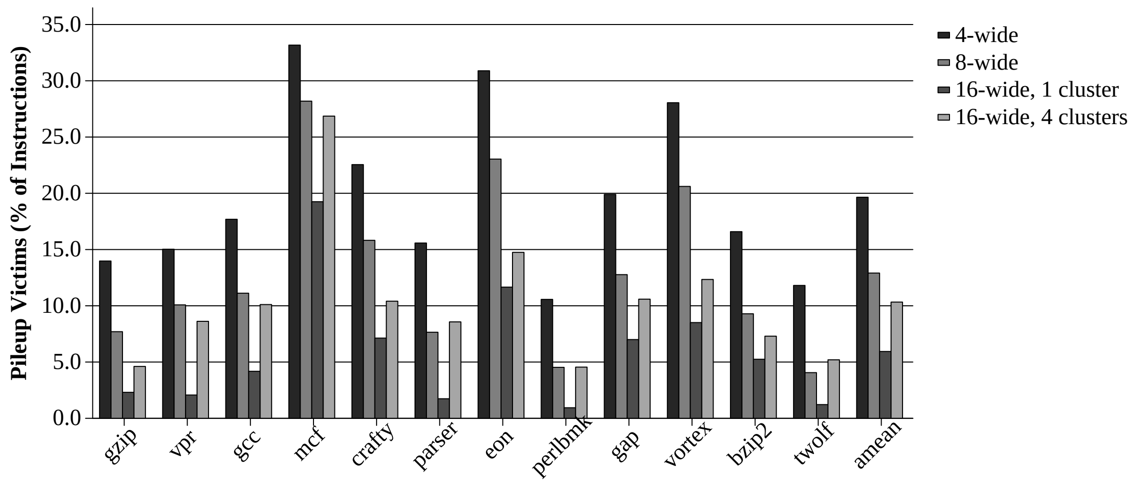


Figure 6.23: Retired Pileup Victims.

these techniques have a very small impact on IPC – less than 0.2% on average for speculative wakeup, and less than 0.5% for select-free scheduling on a 16-wide unclustered machine. These techniques allow the clock frequency to be increased, thereby obtaining higher performance. Both techniques exploit the fact that scheduling time is highly predictable when the prediction is made just a few cycles in advance. On the event of a misspeculation, Select-Free Scheduling uses the existing replay mechanism to reschedule instructions. With Speculative Wakeup, rescheduling happens automatically because the parent tags are used to confirm that instructions are actually ready for execution.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Critical loops are a hindrance to achieving both high clock rate and high IPC because they limit the latency between processing dependent instructions. In the future, as execution width increases, critical loops will become even bigger bottlenecks to high performance. By breaking critical loops in wide-issue machines, ILP can be exploited while at the same time computing chains of dependent instructions quickly. This dissertation evaluates several techniques for breaking the critical execute-bypass, steering, and scheduling loops.

Clustering reduces the latency of both the scheduling and execute-bypass loops. Partitioning a 16-wide execution core into four clusters reduces the latency of the scheduling window by 50% and the latency of the register file with Register Write Specialization by 41%. When using a high-performance steering algorithm (DEP+lsp) with 8 issue ports per cluster, there is less than 10% reduction in IPC. This beats the alternative of pipelining the execute-bypass loop over two cycles, which was shown in Chapter 1 to have a 15% degradation in IPC (and no reduction in latency). The reduction in scheduling window latency is a reduction in the wakeup and tag broadcast portions of the scheduling loop. The latency of the execute-bypass loop is reduced by limiting the number of functional units which receive bypassed values within one cycle. Another advantage of clustering is reduced power consumption. A conventional clustering paradigm reduces power consumption by 19%.

Chapter 4 introduced a new clustering paradigm called *Selective Cluster Receipt* that reduced power by about 5% as compared to the traditional clustered machine while having less than 0.5% impact in IPC. This is because there are 66% fewer fewer register file writes when using SCR. When using SCR, power consumption of a clustered core can be reduced by 25% as compared to the machine with a unified core because of the reduced area of the scheduling windows and register files. In the future, execution cores will have even more clusters. Selective Cluster Receipt will then allow an even greater reduction in power because the fraction of register file writes that are eliminated will be even larger.

Chapter 4 also compared the baseline clustered paradigm and Selective Cluster Receipt against a clustered paradigm with a partitioned register file. The model with a partitioned register file had 5% lower IPC because of the arbitration required to send data between clusters. Although it had lower power consumption in the register file and scheduling windows, it had greater power consumption overall because of the increased complexity in the front-end of the pipeline.

Chapter 5 showed how to implement high-performance steering algorithms that are scalable. By using Virtual Cluster Assignments with trace-cache-based steering algorithms, the number of issue ports per cluster can be increased, allowing for up to a 5% improvement in IPC. As the forwarding delays between clusters increase, the IPC gain from using Virtual Cluster Assignments will increase even more because virtual cluster assignments allow the number of issue ports to be increased while keeping the load balanced across clusters. This increases IPC because it reduces the number of inter-cluster forwarding operations. Out-of-order steering algorithms yield modest improvements in IPC (2%) as compared to other front-end algorithms, but these improvements, too, will increase as the number of clusters increases and the inter-cluster forwarding penalty increases.

Chapter 6 demonstrated that it is possible to break the critical scheduling loop over two or more cycles while still executing dependent instructions in back-to-back cycles. When using speculative wakeup in a 16-wide machine, IPC was reduced by 0.2%. With

Select-Free scheduling, the select logic is removed from the critical scheduling loop. This technique had virtually no effect (less than 0.2%) on IPC. These definitely beat the alternative of pipelining a conventional scheduling loop over two cycles, which was shown in Chapter 1 to have a 10% degradation in IPC.

These techniques have demonstrated that while maintaining a given IPC (or a small reduction in IPC coupled with latency and/or power savings, as is the case for clustering), the latency of the critical loops in the execution core can be reduced. Once critical loops are smaller, either the clock frequency could be increased, or additional resources such as more scheduling window entries or functional units, could be added to fill the clock cycle.

By breaking critical loops, several possible improvements that improve performance and/or reduce power could be made: (1) the size of the out-of-order scheduling window can be increased, which can allow more ILP to be exploited and increase IPC; (2) by reducing critical loop latency, clock frequency can be increased without impacting IPC; (3) power can be reduced by using smaller structures or slower transistors in the logic comprising the critical loop. This dissertation has demonstrated that breaking critical loops can have all of these advantages.

7.2 Future Work

7.2.1 Improvements to Wattch

Several improvements could be made to the power model to more accurately measure static power consumption. In the Wattch model, static power consumption is modeled as a fraction of maximum dynamic power consumption. Wattch calculates maximum dynamic power consumption as a function of wire capacitance (as well as frequency, supply voltage, and transistor sizing), which has little influence on leakage power. A set of functions for computing leakage power could be added to the model. Leakage power can be computed as a function of the number of transistors in a block and threshold voltages and sizes of the transistors used in the block. The number of transistors in a block can be computed using the same basic building block parameters used by the existing Wattch functions. The transistor sizes and threshold voltages can be given as part of the physical design parameters in a header file.

7.2.2 Speculative Inter-Cluster Broadcast

The purpose of Selective Cluster Receipt is to reduce register file power without adding the latency of arbitration to the tag and data communication between clusters. By combining select-free scheduling with clustering, it is possible to reduce the number of register file ports without adding arbitration to the communication path. With this technique, all instructions broadcast their tags to other clusters (possibly using a matrix-style scheduler), but instructions *speculate* that they will broadcast their data N cycles later (where N is the number of cycles between scheduling and execution). During those N cycles, an arbitration mechanism can select a subset of the data values to be broadcast to a particular cluster. This technique would still add some latency to the critical path of the data bypasses: the data path through a 4:1 mux would be added to the critical path. The control for this mux could be set up ahead of time. However, the added latency of this mux can be compensated for by reducing the number of write ports to the register file. Like Selective Cluster Receipt, no arbitration mechanism is needed to reduce the number of data values that are broadcast to each cluster. However, this mechanism would allow the number of write ports for each physical register file to be reduced without adding arbitration to the scheduling latency.

7.2.3 Increasing Window Size with Speculative Scheduling

Speculative Scheduling could be used to increase the scheduling window size as well as reduce its latency. Section 6.1.4 described how Last-Source Prediction can be used to reduce the number of tags in each scheduling entry. A similar technique called Tag Elimination published by Ernst and Austin [21] could be combined with the existing mechanism for further reduction in the number of tags needed for each entry. By reducing the area of each entry, more entries could be added, effectively increasing the size of the scheduling window.

7.2.4 Additional Speculative Scheduling Implementations

Speculative Scheduling can be combined with Select-Free Scheduling or some of the techniques used to implement Select-Free Scheduling for further simplification of the wakeup logic. For example, rather than using parent tags in the wakeup logic to confirm when an instruction is really ready to execute, the availability of source operands could be checked in a scoreboard after instructions have been selected for execution, as is done with Select-Free Scheduling. In this case, Speculative Scheduling would have pileups and collisions like Select-Free Scheduling. Unlike Select-Free Scheduling, however, although there would still be a limit on the number of instructions selected for execution each cycle. If this technique is implemented using array-style wakeup logic, then Speculative Scheduling could be combined with Select-Free scheduling to pipeline the scheduling logic over even more cycles.

7.2.5 Future Work on Steering Algorithms

Additional improvements in steering algorithms can be made for clustered machines that use partitioned Level-1 data caches. For example, the steering algorithms described in this dissertation could be combined with the memory steering heuristics used by

Racunas and Patt [57]. The partitioned cache described in their work can further reduce load latency as compared to the replicated, reduced-port cache used in this dissertation.

Another potential area of improvement for steering algorithms is to increase the scope data dependence graph seen by the steering logic. Front-end in-order steering algorithms do not have knowledge of the future data dependence graph beyond the instruction that is currently being steered. The out-of-order steering algorithms presented in Section 5.2.3 have limited knowledge of the future data dependence graph, but the scope is still limited to one fetch packet. Several techniques could be used to see farther ahead in the dependence graph. First, the compiler has partial knowledge of how and when an instruction's result will be used, although the compiler's scope is limited by control flows not known at compile time. Second, dynamic compilation techniques such as the rePLay Framework [54] can be used to see farther into the data dependence graph. Finally, token-passing techniques such as those used for critical-path predictors [26] can be used to predict when and where register values will be used in the future. Any of these techniques could be implemented to use future dependence graph information to aid in steering decisions. For example, if an instruction is known to have no dependants, it could be steered to any cluster. If the instruction is known to have several dependants, it could be steered to a cluster where there is likely to be room for the dependants as well. As a second example, if an instruction has two parents, both parents could be steered to the same cluster.

Another steering area that has potential is to incorporate a predicted schedule time into the steering algorithm. Note this does not have to be the same as instruction prescheduling (see Section 2.4.2), which adds latency to the front-end of the pipeline. Using techniques such as wakeup prediction [19] as part of the steering heuristic could give front-end steering algorithms insight as to how instructions could be steered in order to minimize functional unit contention.

Bibliography

[1] Algirdas Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, EC-10(9):389–400, September 1961.

[2] Daniel W. Bailey and Bradley J. Benschneider. Clocking design and analysis for a 600-MHz Alpha microprocessor. *IEEE Journal of Solid-State Circuits*, 33(11):1627–1633, November 1998.

[3] Amirali Baniasadi and Andres Moshovos. Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture,* pages 337–347, December 2000.

[4] Ravi Bhargava and Lizy K. John. Improving dynamic cluster assignment for clustered trace cache processors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 264–274, June 2003.

[5] M. Borah, R. M. Owens, and M. J. Irwin. Transistor sizing for low power CMOS circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(9):665–671, June 1996.

[6] Edward Brekelbaum, Jeff Rupley II, Chris Wilkerson, and Bryan Black. Hierarchical scheduling windows. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 27–36, November 2002.

[7] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International*

Symposium on Computer Architecture, pages 83–94, June 2000.

[8] Mary D. Brown and Yale N. Patt. Using internal redundant representations and limited bypass to support pipelined adders and register files. In *Proceedings of the Eighth IEEE International Symposium on High Performance Computer Architecture*, pages 289–298, February 2002.

[9] Mary D. Brown and Yale N. Patt. Demand-only broadcast: Reducing register file and bypass power in clustered execution cores. Technical Report TR-HPS-2004-001, HPS Research Group, The University of Texas at Austin, May 2004.

[10] Mary D. Brown and Yale N. Patt. Demand-only broadcast: Reducing register file and bypass power in clustered execution cores. In *Proceedings of the First Watson Conference on Interaction between Architecture, Circuits, and Compilers, Yorktown Heights, NY*, October 2004.

[11] Doug Burger, Todd Austin, and Steve Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report 1308, University of Wisconsin - Madison Technical Report, July 1996.

[12] Alper Buyuktosunoglu, Ali El-Moursy, and David H. Albonesi. An oldest-first selection logic implementation for non-compacting issue queues. In *15th International ASIC/SOC Conference*, pages 31–35, September 2002.

[13] Ramon Canal and Antonio González. A low-complexity issue logic. In *Proceedings of the 2000 International Conference on Supercomputing*, pages 327–335, May 2000.

[14] Ramon Canal and Antonio González. Reducing the complexity of the issue logic. In *Proceedings of the 2001 International Conference on Supercomputing*, pages 312–320, 2001.

[15] Ramon Canal, Joan-Manuel Parcerisa, and Antonio González. A cost-effective clustered architecture. In *Proceedings of the 1999 ACM/IEEE International Conference on Parallel Architectures and Compilation Techniques*, pages 160–168, October 1999.

[16] Ramon Canal, Joan-Manuel Parcerisa, and Antonio González. Dynamic cluster assignment mechanisms. In *Proceedings of the Sixth IEEE International Symposium on High Performance Computer Architecture*, pages 133–142, February 2000.

[17] R.S. Chappell, P.B. Racunas, Francis Tseng, S.P. Kim, M.D. Brown, Onur Mutlu, Hyesoon Kim, and M.K. Qureshi. The scarab microarchitectural simulator. Unpublished documentation.

[18] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 142–153, June 1998.

[19] Todd E. Ehrhart and Sanjay J. Patel. Reducing the scheduling critical cycle using wakeup prediction. In *Proceedings of the Tenth IEEE International Symposium on High Performance Computer Architecture*, 2004.

[20] M. D. Ercegovac. On-line arithmetic: An overview. *SPIE Real-Time Signal Processing VII*, 495:86–93, August 1984.

[21] Dan Ernst and Todd Austin. Efficient dynamic scheduling through tag elimination. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 37–46, May 2002.

[22] Dan Ernst, Andrew Hamel, and Todd Austin. Cyclone: A broadcast-free dynamic instruction scheduler with selective replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 253–262, 2003.

[23] Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 149–159, December 1997.

[24] Keith I. Farkas, Norman P. Jouppi, and Paul Chow. Register file design considerations in dynamically scheduled processors. In *Proceedings of the Fourth IEEE International Symposium on High Performance Computer Architecture*, pages 40–51, 1998.

[25] Brian Fields, Rastislav Bodík, and Mark D. Hill. Slack: maximizing performance under technological constraints. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 47–58, May 2002.

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

[27] Joseph A. Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 140–150, 1983.

[28] Daniel H. Friendly, Sanjay J. Patel, and Yale N. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 173–181, November 1998.

[29] Bruce A. Gieseke, Randy L. Allmon, Daniel W. Bailey, Bradley J. Benschneider, Sharon M. Britton, John D. Clouser, Harry R. Fair III, James A. Farrell, Michael K.

Gowan, Christopher L. Houghton, James B. Keller, Thomas H. Lee, Daniel L. Leibholz, Susan C. Lowell, Mark D. Matson, Richard J. Matthew, Victor Peng, Michael D. Quinn, Donald A. Priore, Michael J. Smith, and Kathryn E. Wilcox. A 600MHz superscalar RISC microprocessor with out-of-order execution. In *1997 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 176–178, February 1997.

[30] Enric Gilbert, Jesus Sanches, and Antonio González. An interleaved cache clustered VLIW processor. In *Proceedings of the 2002 International Conference on Supercomputing*, pages 210–219, 2002.

[31] Andy Glew. Processor with architecture for improved pipelining of arithmetic instructions by forwarding redundant intermediate data forms. U.S. Patent Number 5,619,664, 1997.

[32] M. Goshima, K. Nishino, Y. Nakashima, S. Mori, T. Kitamura, and S. Tomita. A high-speed dynamic instruction scheduling scheme for superscalar processors. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 225–236, December 2001.

[33] Greg F. Grohoski. Machine organization the IBM RISC system/6000 processor. *IBM Journal of Research and Development*, 34:72–84, 1990.

[34] Dana S. Henry, Bradley C. Kuszmaul, Gabriel H. Loh, and Rahul Sami. Circuits for wide-window superscalar processors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 236–247, June 2000.

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

[36] M.S. Hrishikesh. *Design of Wide-Issue High-Frequency Processors in Wire Delay Dominated Technologies*. Doctoral Dissertation, The University of Texas at Austin, 2004.

[37] M.S. Hrishikesh, Norman P. Jouppi, Keith I. Farkas, Doug Burger, Stephen W. Keckler, and Premkishore Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 14–24, 2002.

[38] Jie S. Hu, N. Vijaykrishnan, and Mary Jane Irwin. Exploring wakeup-free instruction scheduling. In *Proceedings of the Tenth IEEE International Symposium on High Performance Computer Architecture*, February 2004.

[39] Michael Huang, Jose Renau, and Josep Torrellas. Energy-efficient hybrid wakeup logic. In *International Sysmposim on Low-Power Electronics and Design*, pages 196–201, August 2002.

[40] Gregory A. Kemp and Manoj Franklin. PEWs: A decentralized dynamic scheduler for ILP processing. In *International Conference on Parallel Processing*, pages 239–246, August 1996.

[41] R. E. Kessler, E. J. McLellan, and D. A. Webb. The alpha 21264 microprocessor architecture. In *Proceedings of the 16th International Conference on Computer Design*, pages 90–95, October 1998.

[42] Alvin R. Lebeck, Tong Li, Eric Rotenberg, Jinson Koppanalil, and Jaidev Patwardhan. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 59–70, May 2002.

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

[44] P. Michaud and A. Seznec. Data-flow prescheduling for large instruction windows in out-of-order processors. In *Proceedings of the Seventh IEEE International Symposium on High Performance Computer Architecture*, pages 27–36, January 2001.

[45] Enric Morancho, José María Llabería, and Àngel Olivé. Recovery mechanism for latency misprediction. In *Proceedings of the 2001 ACM/IEEE International Conference on Parallel Architectures and Compilation Techniques*, pages 118–128, September 2001.

[46] Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 181–193, June 1997.

[47] Andreas Moshovos and Gurindar S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 235–245, December 1997.

[48] Soner Önder. *Scalable Instruction Processing*. Dissertation, The University of Pittsburgh, 1999.

[49] Soner Önder and Rajiv Gupta. Superscalar execution with dynamic data forwarding. In *Proceedings of the ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, pages 130–135, October 1998.

[50] S. Palacharla, N. P. Jouppi, and J. E. Smith. Quantifying the complexity of superscalar processors. Technical Report TR-96-1328, Computer Sciences Department, University of Wisconsin - Madison, November 1996.

[51] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[52] Joan-Manuel Parcerisa and Antonio González. Reducing wire delay penalty through value prediction. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 317–326, December 2000.

[53] Joan-Manuel Parcerisa, Julio Sahuquillo, Antonio González, and José Duato. Efficient interconnects for clustered microarchitectures. In *Proceedings of the 2002 ACM/IEEE International Conference on Parallel Architectures and Compilation Techniques*, pages 291–300, September 2002.

[54] Sanjay J. Patel, Tony Tung, Satarupa Bose, and Matthew M. Crum. Increasing the size of atomic instruction blocks using control flow assertions. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 303–313, December 2000.

[55] Yale Patt, W. Hwu, and Michael Shebanow. HPS, a new microarchitecture: Rationale and introduction. In *Proceedings of the 18th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 103–107, December 1985.

[56] Yale N. Patt, Steven W. Melvin, W. Hwu, and Michael C. Shebanow. Critical issues regarding HPS, a high performance microarchitecture. In *Proceedings of the 18th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 109–116, 1985.

[57] Paul Racunas and Yale N. Patt. Partitioned first-level cache design for clustered microarchitectures. In *Proceedings of the 17th Annual International Conference on Supercomputing*, pages 22–31, June 2003.

[58] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *The Journal of Supercomputing*, 7:9–50, 1993.

[59] Eric Rotenberg, Quinn Jacobsen, Yiannakis Sazeides, and James E. Smith. Trace processors. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

[60] S. Subramanya Sastry, Subbarao Palacharla, and James E. Smith. Exploiting idle floating-point resources for integer execution. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 118–129, 1998.

[61] André Seznec, Eric Toullec, and Olivier Rochecouste. Register write specialization register read specialization: A path to complexity-effective wide-issue superscalar processors. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 383–394, 2002.

[62] Premkishore Shivakumar and Norman P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical report, Compaq Western Research Laboratory, August 2001.

[63] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[64] Jared Stark, Mary D. Brown, and Yale N. Patt. On pipelining dynamic instruction scheduling logic. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, 2000.

[65] Itsujiro Arita Toshinori Sato. Simplifying wakeup logic in superscalar processors. In *Proceedings of the Euromicro Symposium on Digital Systems Design*, pages 341–346, September 2002.

[66] Gary Scott Tyson and Todd M. Austin. Improving the accuracy and performance of memory communication through renaming. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture,* pages 218–227, December 1997.

[67] T. N. Vijaykumar and Gurindar S. Sohi. Task selection for a multiscalar processor. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, 1998.

[68] Shlomo Weiss and James E. Smith. Instruction issue logic in pipelined supercomputers. *IEEE Transactions on Computers*, C-33(11):1013–1022, November 1984.

[69] Kenneth C. Yeager. The MIPS R10000 superscalar microprocessor. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 28–41, December 1996.

[70] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive branch prediction. In *Proceedings of the 24th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 51–61, November 1991.

[71] R. Zimmermann and W. Fichtner. Low-power logic styles: CMOS versus pass-transistor logic. *IEEE Journal of Solid-State Circuits*, 92(7):1079–1090, July 1997.

[72] Victor V. Zyuban and Peter M. Kogge. The energy complexity of register files. In *Proceedings of the 1998 International Symposium on Low Power Electronic Design*, pages 305–310, August 1998.

[73] Victor V. Zyuban and Peter M. Kogge. Inherently low-power high-performance superscalar architectures. *IEEE Transactions on Computers*, 50(3):268–286, March 2001.

Vita

Mary Douglass Brown was born in Lee County, Alabama on October 3, 1974, the daughter of Dr. Jack B. Brown and Jane Isenhower Brown. She received the Bachelor of Arts degree in Music and the Bachelor of Science degree in Computer Science from Florida State University in 1997. The following year she entered the Ph.D. program at the University of Michigan where she began working with her Ph.D. advisor Dr. Yale N. Patt. She received the Master of Science degree in Computer Science and Engineering in 1999. In the fall of 1999, she followed her advisor to The University of Texas where she continued her Ph.D. studies.

While in graduate school, she served as a teaching assistant for two semesters at The University of Michigan and one semester at The University of Texas. She had summer internships at Motorola, IBM, and Intel. She has published papers in The International Symposium on Microarchitecture (MICRO-33 and MICRO-34), The International Symposium on High Performance Computer Architecture (HPCA-8), and The First Watson Conference on Interaction between Architecture, Circuits, and Compilers (PAC-2). Her graduate studies were supported in part by a University of Michigan Rackham Engineering Fellowship, an IBM Ph.D. Fellowship, and a University of Texas Graduate Engineering Fellowship.

Permanent address: 2600 Lake Austin Blvd, Apt. 4302
Austin, Texas 78703

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.