The Dissertation Committee for Hrishikesh Sathyavasu Murukkathampoondi certifies that this is the approved version of the following dissertation:

Design of Wide-Issue High-Frequency Processors in Wire Delay Dominated Technologies

Committee:

Douglas C. Burger, Supervisor

Craig M. Chase, Supervisor

Lizy K. John

Norman P. Jouppi

Stephen W. Keckler

Yale N. Patt

Design of Wide-Issue High-Frequency Processors in Wire Delay Dominated Technologies

by

Hrishikesh Sathyavasu Murukkathampoondi, B.E., 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 2004

Dedicated to my parents.

Acknowledgments

Several people supported and encouraged me over the course of my time in graduate school and this work would not have been possible without their assistance. I would like to thank my advisor, Dr. Doug Burger, who played a prominent role in educating me and training me to be a good researcher. I am grateful to Dr. Craig Chase, my co-advisor, for his support and guidance. I have benefited greatly from my interactions with Dr. Stephen Keckler. I thank him for sharing his time and knowledge. Thanks to Dr. Lizy John for her support and advice during my first years in graduate school.

Raj Desikan wrote the original processor simulation tool that I used for my research. I would like to thank him for answering my numerous questions. I also thank Vikas Agarwal and Heather Hanson for their help in my understanding of technology related issues. My fellow research students at the Computer Architecture and Technology Laboratory have been important to my development. I gratefully acknowledge them for sharing their knowledge and friendship.

Kartik Agaram, Changkyu Kim, Ramadass Nagarajan, Karthikeyan Sankaralingam, Lakshminarasimhan Sethumadhavan, and Premkishore Shivakumar have been my house mates, travel companions, and close friends over the last five years. Graduate school would not have been so much fun without

the company of this interesting bunch.

Finally, I would like to thank my parents for providing me with the education that allowed me to reach this far. This work would not have been possible without their constant encouragement and unwavering support.

Design of Wide-Issue High-Frequency Processors in Wire Delay Dominated Technologies

Publication No. _____

Hrishikesh Sathyavasu Murukkathampoondi, Ph.D.
The University of Texas at Austin, 2004

Supervisors: Douglas C. Burger
Craig M. Chase

Processor designers have increased performance by improving two parameters — clock frequency and the number of instructions processed per cycle (IPC). Increases in processor pipeline depth has been one factor that has contributed to clock frequency improvements. This research shows that improvements to clock frequency from increasing processor pipeline depths are reaching a point of diminishing returns. It may be possible to continue increasing pipeline depths at the cost of significant increases in design effort and complexity. However, such effort would better spent if directed at increasing IPC.

To increase IPC processors will have to issue more instructions every cycle. Also, the capacity and the number of ports of on-chip structures must

be increased to support wider issue widths. However, large multi-ported structures will have long access latencies that will not scale with technology. Such structures cannot be clocked at aggressive frequencies. A natural solution to the problem of increasing circuit complexity is to partition the architecture into clusters. While clustering reduces the complexity of on-chip structures it introduces other bottlenecks and inefficiencies. These bottlenecks reduce the IPC of a clustered machine compared to an un-clustered machine. The goal of this research is to improve the IPC of a clustered processor to be closer to that of the un-clustered machine. For this purpose we propose new architectural techniques that mitigate the effect of the bottlenecks in clustered processors.

This research proposes a new mechanism, called consumer requested forwarding (CRF) that replaces transfer instructions with hardware signals. In the CRF method, consumer instructions that require values from remote clusters explicitly request the value to be forwarded. This technique significantly reduces the transfer instruction bottleneck. We also propose and evaluate three dynamic instruction steering mechanisms—memory instruction steering, critical operand steering and issue-balance steering. Our studies show that the memory instruction steering policy provides significant IPC improvements over the baseline mechanisms that we evaluated.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Designers have improved microprocessor performance by increasing the number of instructions that are concurrently executed and by increasing clock frequency. Several novel architectural ideas have been developed to improve the instructions per cycle (IPC) that can be processed by the computer. These include branch prediction, data caching, out-of-order issue and execution etc. Improvements in clock frequency have been achieved by developing better semiconductor process technology, faster logic circuits, and aggressively pipelining the architecture. For example, over the past twelve years process technology has been scaled from 1000nm to 130nm. Similarly processor pipeline depths have been increased from 5 stage to 30 stages.

However, recent studies indicate that it will become increasingly difficult to improve processor performance in the future. Both components of performance improvement—clock frequency and IPC—face emerging technology-driven challenges. Recent studies on processor pipeline scaling show that clock frequency improvements from increasing pipeline depths are reaching a point of diminishing return. In the future, designers will have to rely on IPC to a greater extent than before to improve performance.

In order to sustain greater IPC, processors will have to issue more instructions every cycle. Also, microarchitectural structures, such as register files and issue windows, must increase in capacity to be able to uncover greater parallelism in the program. But, technology scaling projections indicate that the latency of large structures with multiple ports will not scale as well as transistor switching speeds. This poor scaling of processor structure latencies will adversely affect IPC. In this dissertation we highlight the challenges facing processor design and propose architectural solutions to address them.

1.1 Pipeline Scaling Trends

Increasing processor pipeline depth is one technique that designers use to improve clock frequency and processor performance. Processor pipelines have grown from a single stage (Intel 8086) to 30 stages (Intel Pentium IV). While increasing pipeline depth does increase clock frequency it reduces the IPC that the processor can sustain.

For example, consider the simple pipeline shown in Figure 1.1. The grey boxes in the diagram represent pipelining overhead while the clear boxes represent useful work. The operating frequency of this pipeline can be increased by decreasing the amount of useful work done every stage and increasing pipeline depth. As illustrated in Figure 1.1, by halving the amount of useful logic per stage we can double the depth of this pipeline (pipeline (b)). Note that, because of latch overhead, doubling the pipeline depth does not double clock frequency. In fact, the new pipeline has added additional latches and increased

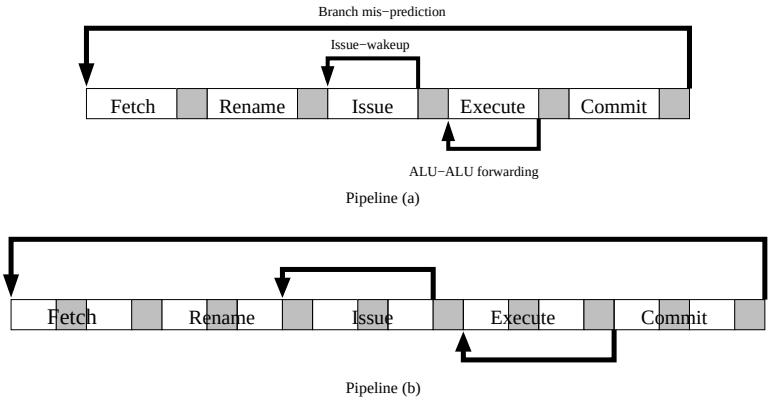

Figure 1.1: A simple processor pipeline with critical loops.

the end to end latency of the pipeline.

If there were no data or control dependences in the instruction stream both these pipelines can commit one instruction every cycle. However, control and data dependences in the instruction stream expose *critical loops* in the pipeline that reduce overall IPC. Figure 1.1 illustrates some of the critical loops in the pipeline. These loops include: issue-wakeup loop, ALU to ALU forwarding, data-cache access and the branch mis-prediction loop. Increasing pipeline depth increases the latency of critical loops due to the additional clocking overhead that is added to the loop. For example, in pipeline (a), when a branch instruction is encountered the address of the next instruction is predicted and instructions are fetched down the speculative path. When the branch instruction reaches the commit stage of the pipeline the prediction is verified. If a mis-prediction is detected the pipeline is flushed and the program counter is set to the correct instruction address. After a mis-prediction no

instructions are committed for several cycles. This branch penalty, for many processors, is equal to the total pipeline depth. Increasing pipeline depth, to increase clock frequency, also increases the latency of critical loops. The deeper pipeline (b) takes more cycles to resolve branch prediction and therefore suffers a larger mis-prediction penalty.

Similar to the branch prediction loop, increasing pipeline depth also increases latency of other critical loops. This increase in critical loop latency, in turn, reduces IPC. There is a tradeoff between increasing pipeline depth (and therefore clock frequency) and IPC. To obtain maximum performance processor designers must balance pipeline depth and IPC. We studied this tradeoff between pipeline depth and IPC. Chapter 2 presents our study in detail. We found that there is an optimal depth beyond which increasing pipeline depths any further will reduce overall performance.

1.2 Process Technology Trends

Process technology has continually fueled processor performance improvements. Every successive technology generation allows greater number of transistors to be packed on chip. Designers utilize the additional transistors in novel ways to improve the number of instructions per cycle (IPC) by building processor enhancements such as more sophisticated branch predictors, on-chip caches, better instruction scheduling mechanisms, structures to support out-of-order execution etc. Reducing feature sizes increases transistor switching speeds and so reduces gate delays. However, projections by the Semiconduc-

tor Industry Association show that technology scaling will cause an increase in wire delay relative to transistor delays [59]. This implies that in the future, even though designers will have a greater number of transistors, the fraction of the chip that can be accessed in a single cycle will decrease.

1.2.1 Wire Delay Scaling

The delay to propagate signals across a wire is determined by three parameters: wire resistance, capacitance and inductance. An increase in wire resistance or capacitance will result in an increase in signal propagation delay. The resistance of the wire is determined by the equation

$$R = \rho * l/A \tag{1.1}$$

where R is the resistance of the wire, ρ is the resistivity of the material, l is the length of the wire, and A is its cross sectional area. As process technology shrinks the *thickness* and *width* of wires is reduced thereby reducing its cross sectional area. This change contributes to an increase in the resistance of the wire. On the other hand, smaller process technology also reduces the length of the wire which in turn reduces wire resistance.

The explanation for wire capacitance is more complicated due to the interaction of multiple conductors close to the wire (i.e. neighboring wires). The overall capacitance of a wire is usually modeled by four parallel-plate capacitors for the top, bottom, and either sides of the wire. The capacitance

of a parallel plate capacitor is determined by the equation

$$C = \epsilon * A/d \tag{1.2}$$

where C is capacitance, ϵ is the permittivity of the material, A is the cross sectional area of the plates, and d is the distance between the plates. As process technology shrinks the wire surface area shrinks (i.e. the area of the plates) contributing to a reduction in the overall capacitance of the wire. However, smaller process technologies also reduce the the spacing between adjacent wires (i.e. distance between the plates) and this change contributes to an increase in overall wire capacitance.

As discussed above, reducing feature sizes results in multiple effects, some that reduce wire resistance and capacitance and some that increase it. The discussion of wire delay presented in this section is a simple, first-order explanation of the factors that contribute to wire resistance and capacitance. Ho *et al.* present a more detailed discussion of wire delay scaling with technology [34]. Their study shows that on-chip wires that scale in length (i.e. shorten as technology scales) have delays that track gate delays. However, the delays of wires that do not scale in length (i.e. global wires) scale poorly relative to transistors.

1.2.2 Implications For Processor Design

Agarwal *et al.* studied the effect of increasing wire delays on on-chip structure latencies and processor performance [2, 3]. They show that because

of increasing wire delays, the latency of memory-oriented microarchitectural structures such as caches and register files increases as technology shrinks. To access such structures in single cycle will require either the capacity of the structures to be decreased or for clock frequencies to be scaled less aggressively.

They quantified the performance effect of scaling a superscalar processor from a 250nm technology to a 35nm technology. In this study they determined the access latencies of on-chip structures at different technology points using a modified version of CACTI [39]. Using a processor simulator, they evaluated the performance of an Alpha 21264-like superscalar processor at different technologies. Their study shows that as technology shrinks the IPC of the processor reduces due to increased structure latencies. Even though overall processor performance improves when it is scaled from 250nm to 35nm this improvement is lower than the increase in clock frequency over the same period. This result implies that even with the increased transistor budget the performance of a monolithic superscalar processor will scale at a rate lower than the clock frequency.

This study shows that the performance of large monolithic processor cores will not scale with technology. Furthermore, future designs that seek to exploit greater instruction level parallelism by issuing more instructions every cycle will require larger on-chip structures with more ports. Such structures cannot operate at high frequencies if designed as monolithic units. Future microarchitectures must be partitioned in order to reduce the access latency of critical structures. Such partitioning will result in a design wherein part of

a structure can be accessed in a single cycle while other parts will require multiple cycles. Though partitioning will enable structures to be clocked at high frequencies it introduces new bottlenecks that still restrict IPC. This dissertation proposes techniques to reduce the bottlenecks in clustered processors.

1.3 Clustered Processors

In clustered superscalar processors on-chip structures such as the instruction issue window, the register file, the data-cache etc are physically partitioned. These partitions, along with functional units, are organized into groups called clusters. Several papers have previously suggested partitioning superscalar processors into clusters [5, 14, 22, 50]. Partitioning a monolithic design into clusters introduces new bottlenecks in the design that degrade the IPC of the clustered processor compared to the monolithic machine. Three factors degrade the IPC of clustered processors compared to a monolithic design. First is the communication delay to access remote operand values and remote memory values. The second factor is poor utilization of processor resources. In clustered processors on-chip resources such as physical registers, functional units, and issue windows are partitioned among clusters and this partitioning reduces overall efficiency. For example, instruction scheduling methods may steer many instructions to one cluster (and fewer instructions to other clusters) resulting in under-utilization of processor resources. The third factor that degrades the IPC of clustered machines is the extra transfer instructions used for inter-cluster communication. These instructions are dynamically generated by

the hardware to transfer operand values between clusters. These instructions do not perform useful computation but they still consume processor resources.

1.4 Thesis Statement

Bottlenecks in clustered processors reduce its IPC compared to that of a monolithic machine. This dissertation proposes techniques to reduce the effects of these bottlenecks and enable a clustered processor to approach the IPC of a monolithic machine.

1.5 Dissertation Contributions

In this dissertation we first explore the scalability of superscalar processor pipelines and show that increasing pipeline depth beyond a point will degrade performance. This performance degradation is because of the increase in latency of certain sections in the pipeline termed *critical loops*. We propose a method to pipeline the issue-wakeup critical loop that will allow the loop to operate in multiple cycles and yet not cause a significant degradation in IPC. This technique, along with methods that address other critical loops, may allow us to increase pipeline depths a little further. But in the longer term designers need to exploit more parallelism in programs to obtain greater performance. Processors will have to issue more instructions every cycle to achieve this goal. Such processors will have to be designed as clusters to reduce the design complexity of individual structures and to enable a high clock frequency.

We examine the bottlenecks in clustered superscalar processors and propose techniques to reduce their effect. We propose two techniques to remove the transfer instruction bottleneck, each of which replace transfer instructions with hardware signals. In the first method, called *consumer requested forwarding* (CRF), inter-cluster dependences are detected when consumer instructions reach the steer stage of the pipeline. Such consumer instructions set a bit in the producer instruction's cluster to forward the value to the consumer. The second method used to perform inter-cluster communication is called *hot-register based forwarding*. This mechanism tracks the registers that are used by each cluster and uses this information to predict where instruction outputs should be forwarded. We found that these techniques of orchestrating inter-cluster communication can almost eliminate all transfer instructions.

We also propose three dynamic instruction steering policies—issue-balance steering, memory steering, and critical-operand steering—to reduce the effect of the inter-cluster communication and cluster resource utilization bottlenecks. The issue-balance steering policy attempts to steer dependent instructions to the same cluster to reduce inter-cluster communication. However, if such an assignment will result in a cluster having more ready instructions than it can issue, instructions are re-assigned to another cluster to avoid stalls due to limited cluster issue-bandwidth. Thus, it attempts to find a balance between inter-cluster communication and processor resource utilization. The critical-operand steering policy attempts to reduce the effect of the inter-cluster communication latency. This policy identifies which of the two source

operands for an instruction is more critical and steers the instruction to the cluster that has fast access to that source operand. Memory-steering works in conjunction with the baseline policies and attempts to reduce the latency of load and store instructions by steering them to clusters with fast access to the data address being accessed. The issue-balance and critical-operand steering policies perform better than two of the three baseline steering mechanisms that we consider. The memory-steering mechanism shows an improvement over all three baseline steering policies.

1.6 Organization

In this chapter we provided a overview of this dissertation work. Chapter 2 examines processor pipeline scaling and shows that performance improvements from pipeline scaling are approaching diminishing returns. The results of this study imply that future processor designs must rely on extracting greater parallelism from the program to increase performance. To improve IPC processor issue widths must increase. Chapter 3 evaluates the SPEC 2000 benchmarks to show that they have considerable instruction level parallelism (ILP) and to determine the on-chip resources required to obtain this ILP. This section also describes our baseline clustered processor in detail. Chapter 4 evaluates the performance of the baseline processor and quantifies the improvement over the baseline if each bottleneck were individually removed. This study shows us the maximum benefit that can be achieved by removing the clustering bottlenecks. In Chapter 5 we propose and evaluate architectural

techniques to remove the transfer instruction bottleneck. The techniques that we propose eliminate almost all transfer instructions. We propose dynamic instruction steering mechanisms in Chapter 6. These steering policies attempt to reduce inter-cluster communication latency and the bottlenecks from poor processor resource utilization. Finally, Chapter 7 presents our concluding remarks.

Chapter 2

Processor Pipeline Scaling

Improvements in microprocessor performance have been sustained by increases in both instruction per cycle (IPC) and clock frequency. Increases in clock frequency have come from technology scaling, improvements in circuit design, and deeper pipelining of designs. In this chapter, we examine how much further reducing the amount of logic per pipeline stage can improve performance.

Reducing the logic per stage, and as a consequence increasing pipeline depth, is a technique to increase clock frequency and therefore overall performance. However, there are certain critical sections in a superscalar processor pipeline that must evaluate in the fewest possible cycles to achieve good performance. These sections are termed *critical loops* [15] and they include—issue wake-up loop, ALU to ALU forwarding, data-cache access and the branch mis-prediction loop. Increasing pipeline depth increases the latency of critical loops due to the additional clocking overhead that is added to the loop. This increase in critical loop latency, in turn, reduces IPC.

There is a tradeoff between increasing pipeline depth (and therefore clock frequency) and IPC. To obtain maximum performance processor de-

signers must balance pipeline depth and IPC. In this chapter we explore this tradeoff by scaling the pipeline depth of an Alpha 21264 processor. The remainder of this chapter is organized in the following manner. To determine the ideal clock frequency we first quantify latch overhead and present a detailed description of this methodology in Section 2.1. Section 2.2 describes the methodology to find the ideal clock frequency, which entails experiments with varied pipeline depths. We present the results of this study in Section 2.3 and discuss its implications. We examine individual critical loops in the pipeline in Section 2.4 and propose a new instruction window design in Section 2.5.

2.1 Estimating Overhead

The clock period of the processor is approximated by the following equation

$$\phi = \phi_{logic} + \phi_{latch} + \phi_{skew} + \phi_{jitter} \tag{2.1}$$

where ϕ is the clock period, ϕ_{logic} is useful work performed by logic circuits, ϕ_{latch} is latch overhead, ϕ_{skew} is clock skew overhead and ϕ_{jitter} is clock jitter overhead. In this section, we describe our methodology for estimating the overhead components, and the resulting values.

A pipelined machine requires data and control signals at each stage to be saved at the end of every cycle. In the subsequent clock cycle this stored information is used by the following stage. Therefore, a portion of each clock period, called *latch overhead*, is required by latches to sample and hold values. Latches may be either edge triggered or level sensitive. Edge-triggered

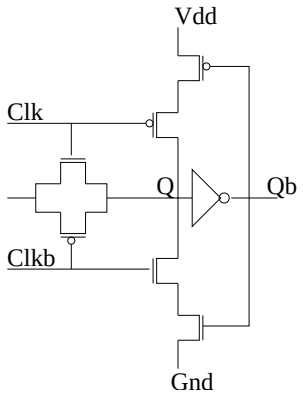

Figure 2.1: Circuit diagram of a basic pulse latch.

latches reduce the possibility of *race through*, enabling simple pipeline designs, but typically incur higher latch overheads. Conversely, level-sensitive latches allow for design optimizations such as "slack-passing" and "time borrowing" [15], techniques that allow a slow stage in the pipeline to meet cycle time requirements by borrowing unused time from a neighboring, faster stage. In this paper we model a level-sensitive pulse latch, since it has low overhead and power consumption [32]. We use SPICE circuit simulations to quantify the latch overhead.

Figure 2.1 shows the circuit for a pulse latch consisting of a transmission gate followed by an inverter and a feed-back path. Data values are sampled and held by the latch as follows. During the period that the clock pulse is high, the transmission gate of the latch is on, and the output of the latch (Q) takes the same value as the input (D). When the clock signal changes to low, the transmission gate is turned off. However, the transistors along one of the

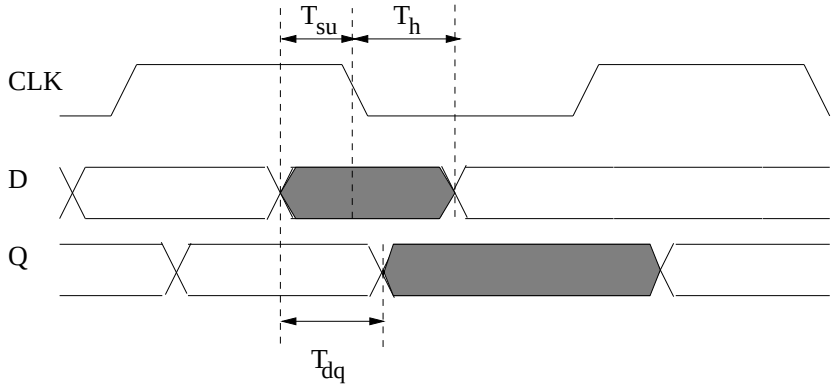

Figure 2.2: Timing diagram of a basic pulse latch. The shaded area indicates that the signal is valid.

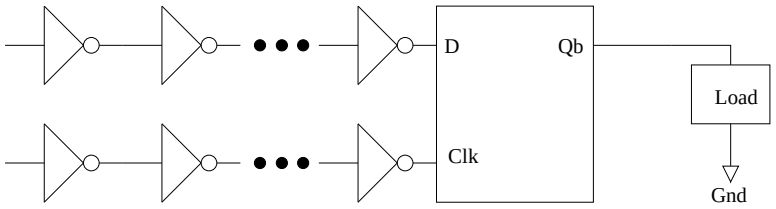

Figure 2.3: Simulation setup to find latch overhead. The clock and data signals are buffered by a series of six inverters and the output drives a similar latch with its transmission gate turned on.

two feedback paths turn on, completing the feedback loop. The inverter and the feedback loop retain the sampled data value until the following clock cycle.

The operation of a latch is governed by three parameters—setup time (T_{su}), hold time (T_h), and propagation delay (T_{dq}), as shown in Figure 2.2. To determine latch overhead, we measured its parameters using the test circuit shown in Figure 2.3. The test circuit consists of a pulse latch with its output driving another similar pulse latch whose transmission gate is turned on. On-

chip data and clock signals may travel through a number of gates before they terminate at a latch. To simulate the same effect, we buffer the clock and data inputs to the latch by a series of six inverters. The clock signal has a 50% duty cycle while the data signal is a simple step function. We simulated transistors at 100nm technology and performed experiments similar to those by Stojanović *et al.* [67], using the same P-transistor to N-transistor ratios. In our experiments, we moved the data signal progressively closer to the falling edge of the clock signal. Eventually when D changes very close to the falling edge of the Clk signal the latch fails to hold the correct value of D. Latch overhead is the smallest of the D-Q delays before this point of failure [67]. We estimated latch overhead to be 36ps (1 FO4) at 100nm technology. Since this delay is determined by the switching speed of transistors, which is expected to scale linearly with technology, its value in FO4 will remain constant at all technologies. Note that the transistor feature sizes we refer to are the drawn gate length as opposed to the effective gate length.

In addition to latch overhead, clock skew and jitter also add to the total overhead of a clock period. A recent study by Kurd *et al.* [43] showed that, by partitioning the chip into multiple clock domains, clock skew can be reduced to less than 20ps and jitter to 35ps. They performed their studies at 180nm, which translates into 0.3 FO4 due to skew and 0.5 FO4 due to jitter. Many components of clock skew and jitter are dependent on the speed of the components, and those that are dependent on the transistor components should scale with technology. However, other terms, such as delay due to

Symbol	Definition	Overhead
ϕ_{latch}	Latch Overhead	1.0 FO4
ϕ_{skew}	Skew Overhead	0.3 FO4
ϕ_{jitter}	Jitter Overhead	0.5 FO4
$\phi_{overhead}$	Total	1.8 FO4

Table 2.1: Overheads due to latch, clock skew and jitter.

process variation, may scale differently, hence affecting the overall scalability. For simplicity we assume that clock skew and jitter will scale linearly with technology and therefore their values in FO4 will remain constant. Table 2.1 shows the values of the different overheads that we use to determine the overall clock period. The total overhead ($\phi_{overhead}$), the sum of latch, clock skew and jitter overhead is equal to 1.8 FO4.

2.2 Pipeline Scaling Methodology

To study the effect of deeper pipelining on performance, we varied the pipeline depth of a modern superscalar architecture similar to the Alpha 21264. This section describes our simulation framework and the methodology we used to perform this study.

We used a simulator developed by Desikan *et al.* that models both the low-level features of the Alpha 21264 processor [20] and the execution core in detail. This simulator has been validated to be within an accuracy of 21% of a Compaq DS-10L workstation. For our experiments, the base latency and capacities of on-chip structures matched those of the Alpha 21264, and

Integer	Floating Point
164.gzip	171.swim
175.vpr	172.mgrid
176.gcc	173.applu
181.mcf	177.mesa
197.parser	178.galgel
252.eon	179.art
253.perlbmk	183.equake
256.bzip2	188.ammp
	189.lucas

Table 2.2: SPEC 2000 benchmarks used in all simulation experiments.

the level-2 cache was configured to be 2MB. The capacities of the integer and floating-point register files were increased to 512 each, so that the performance of deep pipelines was not unduly constrained due to unavailability of registers. We modified the execution core of the simulator to permit the addition of more stages to different parts of the pipeline. The modifications allowed us to vary the pipeline depth of different parts of the processor pipeline, including the execution stage, the register read stage, the issue stage, and the commit stage.

Table 2.2 lists the benchmarks that we simulated for our experiments, which include integer and floating-point benchmarks taken from the SPEC 2000 suite. All experiments skip the first 500 million instructions of each benchmark and simulate the next 500 million instructions.

We use Cacti to model on-chip microarchitectural structures and to estimate their access times [39]. All major microarchitectural structures—data cache, register file, branch predictor, register rename table and instruction

issue window—were modeled at 100nm technology and their capacities and configurations were chosen to match the corresponding structures in the Alpha 21264. We use the latencies of the structures obtained from Cacti to compute their access penalties (in cycles) at different clock frequencies.

2.3 Optimal Pipeline Depth

We find the clock frequency that will provide maximum performance by simulating processor pipelines clocked at different frequencies. The clock period of the processor is determined by the following equation: $\phi = \phi_{logic} + \phi_{overhead}$. The overhead term is held constant at 1.8 FO4, as discussed in Section 2.1. We vary the clock frequency ($1/\phi$) by varying ϕ_{logic} from 2 FO4 to 16 FO4. The number of pipeline stages (clock cycles) required to access an on-chip structure, at each clock frequency, is determined by dividing the access time of the structure by the corresponding ϕ_{logic}. For example, if the access time of the level-1 cache at 100nm technology is 0.28ns (8 FO4), for a pipeline where ϕ_{logic} equals 2 FO4 (0.07ns), the cache can be accessed in 4 cycles.

Though we use a 100nm technology in this study, the access latencies at other technologies in terms of the FO4 metric will remain largely unchanged at each corresponding clock frequency, since delays measured in this metric are technology independent. Table 2.3 shows the access latencies of structures at each ϕ_{logic}. These access latencies were determined by dividing the structure latencies (in pico seconds) obtained from the cacti model by the corresponding

ϕ_{logic} (FO4)	DL1	Branch Predictor	Rename Table	Issue Window	Register File
2	16	10	9	9	6
3	11	7	6	6	4
4	9	5	5	5	3
5	7	4	4	4	3
6	6	4	3	3	2
7	6	3	3	3	2
8	5	3	3	3	2
9	5	3	2	2	2
10	4	2	2	2	2
11	4	2	2	2	1
12	4	2	2	2	1
13	4	2	2	2	1
14	4	2	2	2	1
15	3	2	2	2	1
16	3	2	2	2	1
Alpha 21264 (17.4)	3	1	1	1	1

Table 2.3: Access latencies (clock cycles) of microarchitectural structures at 100nm technology (drawn gate length). The last row shows the latency of on-chip structures on the Alpha 21264 processor (180nm).

clock period. Table 2.4 shows the latencies for various integer and floating-point operations at different clocks. To compute these latencies we determined ϕ_{logic} for the Alpha 21264 processor (800MHz, 180nm) by attributing 10% of its clock period to latch overhead (approximately 1.8 FO4). Using this ϕ_{logic} and the functional unit execution times of the Alpha 21264 (in cycles) we computed the execution latencies at various clock frequencies. In all our simulations, we assumed that results produced by the functional units can be fully bypassed to any stage between Issue and Execute.

ϕ_{logic} (FO4)	Integer		Floating Point			
	Add	Mult	Add	Div	Sqrt	Mult
2	9	61	35	105	157	35
3	6	41	24	70	105	24
4	5	31	18	53	79	18
5	4	25	14	42	63	14
6	3	21	12	35	53	12
7	3	18	10	30	45	10
8	3	16	9	27	40	9
9	2	14	8	24	35	8
10	2	13	7	21	32	7
11	2	12	7	19	29	7
12	2	11	6	18	27	6
13	2	10	6	17	25	6
14	2	9	5	15	23	5
15	2	9	5	14	21	5
16	2	8	5	14	20	5
Alpha 21264 (17.4)	1	7	4	12	18	4

Table 2.4: Execution latencies (clock cycles) of integer and floating-point operations at 100nm technology (drawn gate length). The functional units are fully pipelined and new instructions can be assigned to them every cycle. The last row shows the execution latency on the Alpha 21264 processor (180nm).

The access latencies of the structures increase as ϕ_{logic} is decreased. In certain cases the access latency remains unchanged despite a change in ϕ_{logic}. For example, the access latency of the register file is 0.39ns at 100nm technology. If ϕ_{logic} were 10 FO4 the access latency of the register file would be approximately 1.1 cycles. Conversely, if ϕ_{logic} was reduced to 6 FO4, the access latency would be 1.8 clock cycles. In both cases the access latency is rounded to 2 cycles.

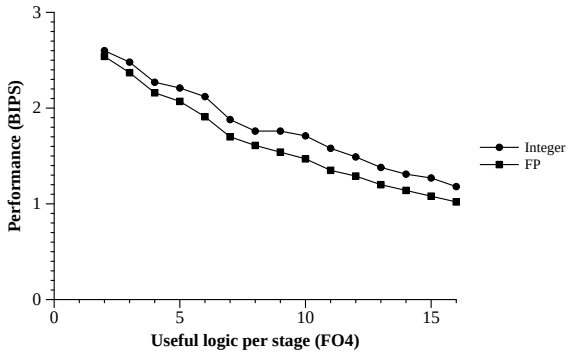

Figure 2.4: The harmonic mean of the performance of integer and floating point benchmarks without latch overhead, clock skew and jitter.

By varying the processor pipeline as described above, we determine how deeply a high-performance design can be pipelined before overheads, due to latch, clock skew and jitter, and reduction in IPC, due to increased on-chip structure access latencies, begin to reduce performance. We consider a processor configuration similar to the Alpha 21264: 4-wide integer issue and 2-wide floating-point issue. We used a modified version of the simulator developed by Desikan *et al.* [20]. Figure 2.4 shows the harmonic mean of the performance of SPEC 2000 benchmarks if there were no overheads associated with pipelining ($\phi_{overhead} = 0$) and performance was inhibited only by control and data dependences in the benchmark. The x-axis in Figure 2.4 represents ϕ_{logic} and the y-axis shows performance in billions of instructions per second (BIPS). Performance was computed as a product of IPC and clock frequency—equal to $(1/\phi_{logic})$. The integer benchmarks show a greater overall performance compared to the floating point benchmarks. One reason for the difference in performance is that some floating point benchmarks have high

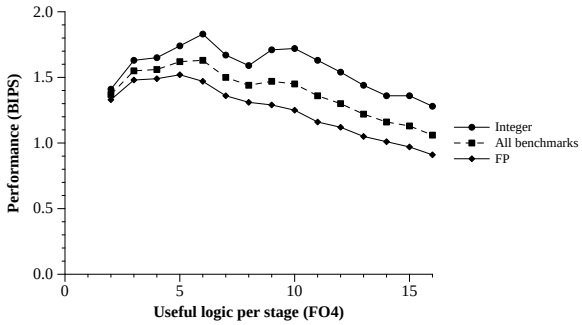

Figure 2.5: The harmonic mean of the performance of integer and floating point benchmarks, executing on an out-of-order pipeline, accounting for latch overhead, clock skew and jitter. For integer benchmarks best performance is obtained with 6 FO4 of useful logic per stage (ϕ_{logic}). For floating-point benchmarks the optimal ϕ_{logic} is 5 FO4.

data cache miss rates which lower their performance. A second reason is that the integer pipeline is wider (4-wide) than the floating point pipeline (2-wide). For both sets of benchmarks, if there was no clock overhead, performance continually increases with increasing pipeline depth. However, doubling the clock frequency does not double performance. For example, when ϕ_{logic} is reduced from 8 to 4 FO4, the ideal improvement in performance is 100%. Between the same two clock points the integer benchmarks show a performance improvement of 29% and the floating-point benchmarks show an improvement of 34%. As ϕ_{logic} is further decreased the improvement in performance deviates further from the ideal value.

Figure 2.5 shows a plot of the performance of SPEC 2000 benchmarks with $\phi_{overhead}$ set to 1.8 FO4. Unlike in Figure 2.4, in this graph the clock frequency is determined by $1/(\phi_{logic} + \phi_{overhead})$. For example, at the point in

the graph where ϕ_{logic} is equal to 8 FO4, the clock frequency is 1/(9.8 FO4). Initially, as ϕ_{logic} is decreased, performance improves due to the increase in clock frequency. Beyond a certain point, the IPC reduction due to critical loops and the clock overhead from additional latches outweigh the gains from clock frequency. Figure 2.5 shows this optimal ϕ_{logic} for integer benchmarks is 6 FO4 and for floating-point benchmarks the optimal ϕ_{logic} is 5 FO4. The dashed curve plots the harmonic mean of both sets of benchmarks and shows the optimal ϕ_{logic} to be 6 FO4.

As mentioned earlier, the access latencies of on-chip structures and execution unit latencies are rounded up to the closest whole number. Therefore, across some of the clock frequency points, shown in Figure 2.5, access and execution latencies remain unchanged. At these points the pipeline is unbalanced. The slope discontinuities in the performance curve are due to such pipeline imbalances.

Critical loops affect the floating-point benchmarks to a lesser extent than the integer benchmarks and therefore their optimal pipeline depth is greater. For example, floating-point programs have fewer branch mispredictions as compared to integer programs and so increasing the misprediction penalty affects them to a lesser extent. Furthermore, we observed that critical loops affect integer benchmarks in a similar fashion and therefore all these programs had the same optimal ϕ_{logic}. Floating-point benchmarks, on the other hand, have optimal ϕ_{logic} points ranging between 4-7 FO4.

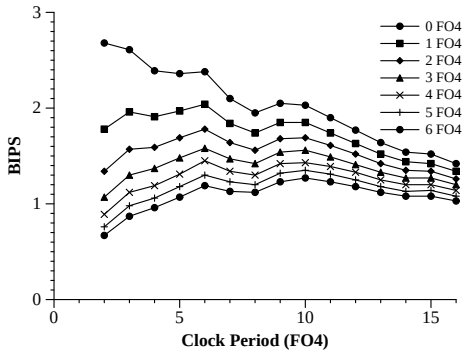

Figure 2.6: The harmonic mean of the performance of integer benchmarks, executing on an out-of-order pipeline for various values of $\phi_{overhead}$.

2.3.1 Sensitivity of ϕ_{logic} to $\phi_{overhead}$

Previous sections assumed that components of $\phi_{overhead}$, such as skew and jitter, would scale with technology and therefore overhead would remain constant. In this section, we examine performance sensitivity to $\phi_{overhead}$. Figure 2.6 shows a plot of the performance of integer SPEC 2000 benchmarks against ϕ_{logic} for different values of $\phi_{overhead}$. In general, if the pipeline depth were held constant (i.e. constant ϕ_{logic}), reducing the value of $\phi_{overhead}$ yields better performance. However, since the overhead is a greater fraction of their clock period, deeper pipelines benefit more from reducing $\phi_{overhead}$ than do shallow pipelines. Interestingly, the optimal value of ϕ_{logic} is fairly insensitive to $\phi_{overhead}$. In section 2.1 we estimated $\phi_{overhead}$ to be 1.8 FO4. Figure 2.6 shows that for $\phi_{overhead}$ values between 1 and 5 FO4 maximum performance is still obtained at a ϕ_{logic} of 6 FO4.

2.3.2 Related Work

Previously, Kunkel and Smith examined the tradeoff between pipeline depth and IPC for a CRAY 1-S supercomputer [42] to determine the number of levels of logic per pipeline stage that provides maximum performance. They assumed the use of Earle latches between stages of the pipeline, which were representative of high-performance latches of that time. They concluded that, in the absence of latch and skew overheads, absolute performance increases as the pipeline is made deeper. But when the overhead is taken into account, performance increases up to a point beyond which increases in pipeline depth reduce performance. They found that four to eight gate levels per pipeline stage yields optimal performance. This result roughly translates to a delay of about 6 FO4 to 11 FO4 in CMOS technology [35]. Out results show that the optimal clock period is between 6 FO4 to 8 FO4. It is interesting to note that despite changes in technology the optimal clock period for processors remains more or less unchanged.

Dubey and Flynn discussed optimal pipelining in an analytical framework [21]. They showed that the optimal pipeline depth decreases with increasing clock overhead. Hartstein and Puzak studied pipeline scalability using an analytical model and with simulations [30]. They developed an analytical model of the processor pipeline whose input parameters include—clock overhead, the total logic delay of the processor, and the degree of superscalar processing. Their analysis shows that there is an optimal point beyond which increasing pipeline depth will decrease performance. They observed that that

modern workloads, written in C++ and Java, have deeper optimal pipeline depths than SPEC workloads. In addition, they also observed that for workloads that have a lot of parallelism (ILP), increasing the issue width of the processor will result in shorter optimal pipeline depths. Sprangle and Carmean studied pipeline depths in the context of a Pentium 4 processor [64]. Their study shows that the pipeline depth of the processor could be doubled before the benefits from clock frequency are overcome by the degradation in IPC.

In our study, we did not consider the effect of power on pipeline scaling. Other research suggests that if processor power is taken into account the optimal ϕ_{logic} is about 18 FO4 [31, 65]. Both these studies focus on the pipeline depth that will maximize a power-performance metric ($BIPS^3/Watt$). However, depending on target performance requirements and the ability of the processor packaging to cool the chip, designers may be willing to tradeoff additional power for performance even if it means straying from the optimal power/performance point. Furthermore, several research efforts that are focused at reducing on-chip power may alter these power-performance tradeoffs [29, 40, 49, 54, 74, 77].

2.4 Effect of Pipelining on IPC

Thus far we have examined scaling of the entire processor pipeline. Increasing overall pipeline depth of a processor decreases IPC because of dependencies within *critical loops* in the pipeline [9]. These critical loops include issuing an instruction and waking its dependent instructions (issue wake up),

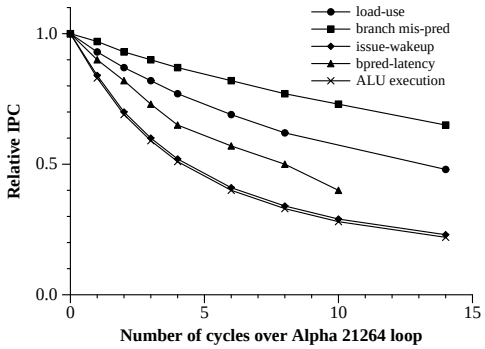

Figure 2.7: IPC sensitivity to critical loops in the data path. The x-axis of this graph shows the number of cycles the loop was extended over its length in the Alpha 21264 pipeline. The y-axis shows relative IPC.

issuing a load instruction and obtaining the correct value (DL1 access time), the latency to predict a branch instruction, predicting a branch and resolving the correct execution path, and the latency to forward operands between ALUs. For high performance it is important that these loops execute in the fewest cycles possible. When the processor pipeline depth is increased, the lengths of these critical loops are also increased, causing a decrease in IPC. In this section we quantify the performance effects of each of the above critical loops and in Section 2.5 we propose a technique to design the instruction window so that in most cases the issue-delay loop is 1 cycle.

To examine the impact of the length of critical loops on IPC, we scaled the length of each loop independently, keeping the access latencies of other structures to be the same as those of the Alpha 21264. Figure 2.7 shows the IPC sensitivity of the integer benchmarks to the branch misprediction penalty, the DL1 access time (load-use), the issue wake-up loop, the ALU

to ALU forwarding loop (ALU execution), and the latency to predict branch instructions (bpred-latency). The x-axis of this graph shows the number of cycles the loop was extended over its length in the Alpha 21264 pipeline. The y-axis shows IPC relative to the baseline Alpha 21264 processor. IPC is most sensitive to the issue wake-up loop and the ALU execution loop, followed by the bpred-latency loop, the load-use and the branch misprediction penalty. The issue wake-up loop is most sensitive because it affects every instruction that is dependent on another instruction for its input values.

The IPC degradation due to each individual loop is a function of how frequently the loop is encountered and the penalty (in cycles) of the loop. Increasing the latency of some of the loops, such as the issue wake-up loop, also increases the branch-misprediction penalty. Furthermore, the branch mis-prediction penalty is paid only on a mis-predicted branch and good branch predictors ensure that this penalty is paid infrequently. Therefore, IPC is less sensitive to branch mis-prediction loop than the other loops. The floating-point benchmarks showed similar trends with regard to their sensitivity to critical loops. However, overall they were less sensitive to all critical loops than integer benchmarks.

The results from Figure 2.7 show that IPC is most sensitive to the latency of the issue wake-up loop and the forwarding. Since these loops affect the ability of the processor to execute dependent instructions back to back it is important that their latency is as low as possible. In the next section we propose a method to reduce the penalty of the issue wake-up loop.

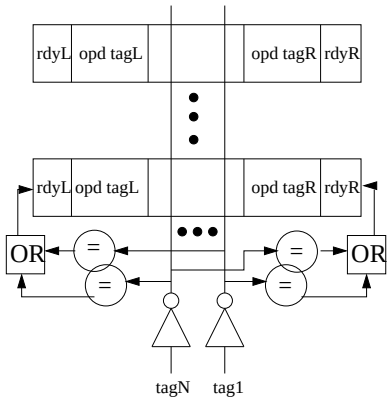

Figure 2.8: A high-level representation of the instruction window.

2.5 A Segmented Instruction Window Design

In modern superscalar pipelines, the instruction issue window is a critical component, and a naive strategy that prevents dependent instructions from being issued back to back would unduly limit performance. In this section we propose a method to pipeline the instruction issue window to enable clocking it at high frequencies.

To issue new instructions every cycle, the instructions in the instruction issue window are examined to determine which ones can be issued (wake up). The instruction selection logic then decides which of the woken instructions can be selected for issue. Stark *et al.* showed that pipelining the instruction window, but sacrificing the ability to execute dependent instructions in consecutive cycles, can degrade performance by up to 27% compared to an ideal machine [66].

Figure 2.8 shows a high-level representation of an instruction window.

Every cycle that a result is produced, the tag associated with the result (*destination tag*) is broadcast to all entries in the instruction window. Each instruction entry in the window compares the destination tag with the tags of its source operands (*source tags*). If the tags match, the corresponding source operand for the matching instruction entry is marked as ready. A separate logic block (not shown in the figure) selects instructions to issue from the pool of ready instructions. At every cycle, instructions in any location in the window can be woken up and selected for issue. In the following cycle, empty slots in the window, from instructions issued in the previous cycle, are reclaimed and up to four new instructions can be written into the window. In this section, we first describe and evaluate a method to pipeline instruction wake up and then evaluate a technique to pipeline instruction selection logic.

2.5.1 Pipelining Instruction Wakeup

Palacharla *et al.* [50] argued that three components constitute the delay to wake up instructions: the delay to broadcast the tags, the delay to perform tag comparisons, and the delay to OR the individual match lines to produce the ready signal. Their studies show that the delay to broadcast the tags will be a significant component of the overall delay as technology shrinks. To reduce the tag broadcast latency, we propose organizing the instruction window into stages, as shown in Figure 2.9. The instruction window is a collapsing window that reclaims slots after an instructions have been issued. Each stage consists of a fixed number of instruction entries and consecutive

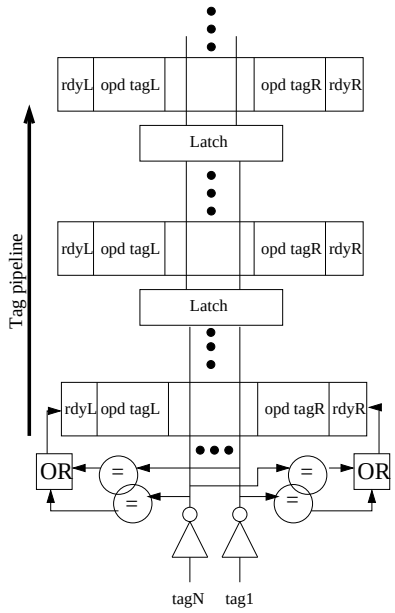

Figure 2.9: A segmented instruction window wherein the tags are broadcast to one stage of the instruction window at a time. We also assume that instructions can be selected from the entire window.

stages are separated by latches. A set of destination tags are broadcast to only one stage during a cycle. The latches between stages hold these tags so that they can be broadcast to the next stage in the following cycle. For example, if an issue window capable of holding 32 instructions is divided into two stages of 16 entries each, a set of tags are broadcast to the first stage in the first cycle. In the second cycle the same set of tags are broadcast to the next stage, while a new set of tags are broadcast to the first 16 entries. At every cycle, the entire instruction window can potentially be woken up by a different set of destination tags at each stage. Since each tag is broadcast across only a small part of the window every cycle, this instruction window can be clocked at high

frequencies. However, the tags of results produced in a cycle can wake up instructions only in the first stage of the window during that cycle. Therefore, dependent instructions can be issued back to back only if they are in the first stage of the window.

We evaluated the effect of pipelining the instruction window on IPC by varying the pipeline depth of a 32-entry instruction window from 1 to 10 stages. Figure 2.10 shows the results from our experiments when the number of stages of the window is varied from 1 to 10. Note that the x-axis on this graph is the pipeline depth of the wake-up logic. The plot shows that IPC of integer and vector benchmarks remain unchanged until the window is pipelined to a depth of 4 stages. The overall decrease in IPC of the integer benchmarks when the pipeline depth of the window is increased from 1 to 10 stages is 22%. The floating-point benchmarks show a decrease of 8% for the same increase in pipeline depth. Note that this decrease is small compared to that of naive pipelining, which prevents dependent instructions from issuing consecutively. We also evaluated a 128-entry instruction window and found that IPC is not reduced until the window is segmented into six stages. For a 6-stage window the IPC drops by 3%.

There are two reasons why pipelining instruction wake up results in a just a small degradation in IPC. About 90% of all instructions selected for issue are selected from Stage 1. Since dependent instructions and their producers are separated by only a few dynamic instructions, in most cases the dependent instruction is also in Stage 1 when the producer is issued. Therefore, most

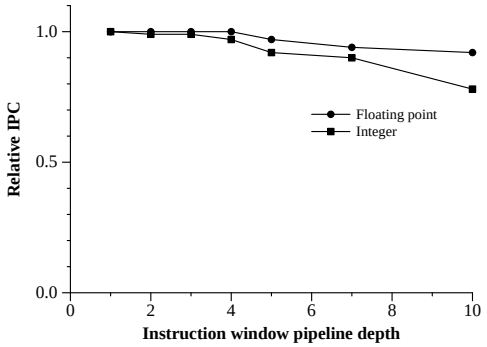

Figure 2.10: IPC sensitivity to instruction window pipeline depth, assuming all entries in the window can be considered for selection.

dependent instructions are woken up in the same cycle that their producers are issued. The second reason for such a small IPC degradation is that we used a an Alpha 21264-like compacting instruction window. The compacting feature of the instruction window also aids in moving dependent instructions physically closer to their producers.

2.5.2 Pipelining Instruction Select

In addition to wake-up logic, the selection logic contributes to the latency of the instruction issue pipeline stage. In a conventional processor, the select logic examines the entire instruction window to select instructions for issue. We propose to decrease the latency of the selection logic by reducing its fan-in. As with the instruction wake-up, the instruction window is partitioned into stages as shown in Figure 2.11. The selection logic is partitioned into two operations: *preselection* and *selection*. A preselection logic block is associated with all stages of the instruction window (S2-S4) except the first one. Each of

these logic blocks examines all instructions in its stage and picks one or more instructions to be considered for selection. A selection logic block (S1) selects instructions for issue from among all ready instructions in the first section and the instructions selected by S2-S4. Each logic block in this partitioned selection scheme examines fewer instructions compared to the selection logic in conventional processors and can therefore operate with a lower latency.

Several configurations of instruction window and selection logic are possible depending on the instruction window capacity, pipeline depth, and selection fan-in. Pipelining instruction wake up into four stages reduces IPC marginally (Section 2.5.1). Therefore, for this study we evaluate the 4-stage implementation shown in Figure 2.11. This instruction window consists of 32-entries partitioned into four stages and is configured so that the fan-in of S1 is 16. Since each stage in the window contains 8 instructions and all the instructions in Stage 1 are considered for selection by S1, up to 8 instructions may be pre-selected. Older instructions in the instruction window are considered to be more critical than younger ones. Therefore the preselection blocks are organized so that the stages that contain the older instructions have a greater share of the pre-selected instructions. The logic blocks S2, S3, and S4 pre-select instructions from the second, third, and fourth stage of the window respectively. Each select logic block can select from any instruction within its stage that is ready. However, S2 can pre-select a maximum of five instructions, S3 a maximum of 2 and S4 can pre-select only one instruction. The selection process works in the following manner. At every clock cycle, preselection logic

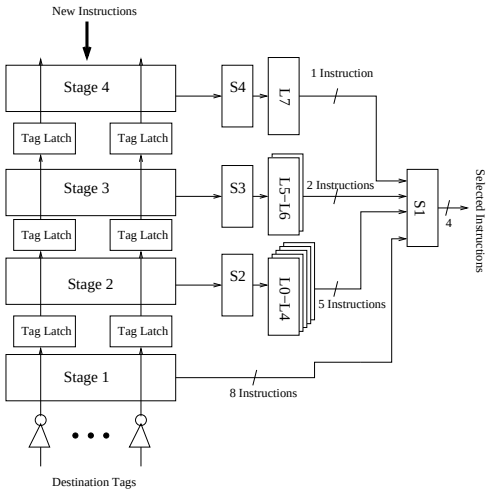

Figure 2.11: A 32-entry instruction window partitioned into four stages with a selection logic fan-in of 16 instructions.

blocks S2-S4 pick from ready instructions in their stage. The instructions preselected by these blocks are stored in latches L1-L7 at the end of the cycle. In the second cycle the select logic block S1 selects 4 instructions from among all the ready instructions in Stage 1 and those in L1-L7 to be issued to functional units.

With an instruction window and selection logic as described above, the IPC of integer benchmarks was reduced by only 4% compared to a processor with a single cycle, 32-entry, non-pipelined instruction window and select fan-in of 32. The IPC of floating-point benchmarks was reduced by only 1%. The rather small impact of pipelining the instruction window on IPC is not surprising. The floating-point benchmarks have fewer dependences in their instruction streams than integer codes, and therefore remain unaffected by

the increased wake up penalties. For the integer benchmarks, most of the dependent instructions are fairly close to the instructions that produce their source values. Also, the instruction window adjusts its contents at the beginning of every cycle so that the older instructions collect to one end of the window. This feature causes dependent instructions to eventually collect at the "bottom" of the window and thus enables them to be woken up with less delay. We evaluated the latency of a 32-entry monolithic issue window and a 32-entry, 4-stage window using an extended version of CACTI [3]. At 90nm technology, the 4-stage window can be clocked at 21% higher frequency compared to the monolithic window. This segmented window design is capable of operating at greater frequencies than conventional designs at the cost of minimal degradation in IPC.

2.5.3 Related Work

Raasch *et al.* proposed a technique to partition the instruction window into segments [55]. Their methods employs a dependence-based mechanism to promote instructions from segment to segment and finally to an issue buffer. Instructions are issued to functional units only from the issue buffer.

Lebeck *et al.* observed that instructions that are waiting for the completion of long latency operations will occupy the issue window for a long period of time [44]. Such instructions cannot be selected for execution until the long latency operation completes. They proposed moving the entire chain of instructions that are dependent on a long latency operation to a buffer called

the *waiting instruction buffer* (WIB). Every cycle, instructions are selected for execution from a 32-entry issue window. The instructions in the WIB are not considered for execution. Once the long latency operation is complete the corresponding dependent instructions from the WIB are reinserted into the issue window.

The partitioned instruction window designs proposed by Raash *et al.* and Lebeck *et al.* are selective about which instructions they consider for issue to functional units. Unlike the segmented issue window that we propose, they do not consider all instructions for issue. So, the above techniques are better suited for designing large issue windows. However, they have greater design complexity compared to the segmented instruction window. Therefore the segmented issue window may be clocked at higher frequencies compared to the other two designs.

Stark *et al.* [66] proposed a technique to pipeline instruction wake up and select logic. In their technique, instructions are woken up "speculatively" when their *grandparents* are issued. The rationale behind this technique is that if an instruction's grandparents' tags are broadcast during the current cycle its *parents* will probably be issued the same cycle. While speculatively woken instructions can be selected, they cannot be issued until their parents have been issued. Although this technique reduces the IPC of the processor compared to a conventional 1-cycle instruction window, it enables the instruction window to function at a higher clock frequency.

Brown *et al.* proposed a method to move selection logic off the critical

path [12]. In this method, wake-up and select are partitioned into two separate stages. In the first stage (wake-up) instructions in the window are woken up by producer tags, similar to a regular instruction window. All instructions that wake up speculate they will be selected for issue in the following cycle and assert their "available" signals. In the next cycle, the result tags of these instructions are broadcast to the window, as though all of them have been issued. However, the selection logic selects only a limited number of instructions from those that asserted their "available" signal. Instructions that do not get selected (*collision victims*) and any dependents that are woken up before they can be issued (*pileup victims*) are detected and re-scheduled. The authors show that this technique has an IPC within 3% of a machine with single-cycle scheduling logic.

The techniques proposed by Stark *et al.* and Brown *et al.* can be used in conjunction with the segmented issue window that we propose. However, both these methods make instruction scheduling decisions speculatively and have complex mechanisms to recover from mis-speculation. These mis-speculations will also result in additional power consumption. The segmented issue window does not schedule instructions speculatively and so it does not require complex recovery logic.

2.6 Summary

In this chapter, we measured the effects of varying clock frequency on the performance of a Alpha 21264 pipeline. We determined the amount of

useful logic per stage (ϕ_{logic}) that will provide the best performance is approximately 6 FO4 inverter delays for integer benchmarks. If ϕ_{logic} is reduced below 6 FO4 the improvement in clock frequency cannot compensate for the decrease in IPC. Conversely, if ϕ_{logic} is increased to more than 6 FO4 the improvement in IPC is not enough to counteract the loss in performance resulting from a lower clock frequency. The clock period ($\phi_{logic} + \phi_{overhead}$) at the optimal point is 7.8 FO4 for integer benchmarks, corresponding to a frequency of 3.6GHz at 100nm technology. This optimal clock frequency can be achieved only if on-chip microarchitectural structures can be pipelined to operate at high frequencies. We identified the instruction issue window as a critical structure, which will be difficult to scale to those frequencies. We propose a segmented instruction window design that will allow it to be pipelined to four stages without significant decrease in IPC.

In studying the scalability of processor pipelines we made several optimistic assumptions. For example, all functional units and on-chip structures are fully pipelined and can accept new requests every cycle. We assumed that results produced by the functional units can be fully bypassed to any stage between Issue and Execute. There are significant design challenges that have to be met to achieve such a high degree of pipelining.

Reducing the penalty of critical loops may enable us to reduce ϕ_{logic} up to and maybe even beyond 6 FO4. We proposed a method to pipeline the instruction issue window and reduce the penalty of the issue-wakeup critical loop. Several other methods have been proposed to address some of the crit-

ical loops in the pipeline data path [10, 12, 55, 66]. While these architectural techniques enable the pipelining of key microarchitectural structures they also increase design complexity. This additional design complexity in itself may restrict pipeline depths and therefore improvement in clock frequency.

Future processor architectures cannot rely on clock frequency alone to improve performance and will have to improve instruction throughput to a greater extent than before. Building wider issue machines to increase IPC will also increase the complexity of design. However, any improvement in IPC directly translates to a corresponding improvement in processor performance. But improving clock frequency by a factor will not improve overall performance by the same factor. Therefore, building wider issue machines could potentially improve processor performance to a greater extent than improving clock frequency alone.

Chapter 3

Wide Issue Processors

Increasing pipeline depth to improve processor performance has proven to be an effective design strategy so far. However, pipeline scaling studies show that performance improvements from deeper pipelines are gradually approaching a point of diminishing returns [30, 35, 64]. Though improvements in process technology will continue to increase clock frequency the bulk of future performance has to be obtained by exploiting greater parallelism in the instruction stream. Future processor designs will require large on-chip structures to dynamically detect independent instructions and must issue a greater number of instructions every cycle. Any future wide-issue design will have to employ partitioning to reduce the complexity and latency of on-chip structures.

In this chapter we first examine the SPEC 2000 benchmarks to quantify how much IPC can be exploited from these workloads and determine the on-chip resources required to obtain this IPC. We then describe, in detail, the baseline architecture that we will use in our studies in later chapters. We also provide a brief overview of previous work on partitioned architectures.

3.1 Instruction Level Parallelism in Programs

Pipeline scaling studies show that clock frequency improvements from increasing pipeline depth will slow down. It is imperative that future processors exploit ILP to greater lengths than before to continue improving processor performance. Postiff *et al.* analyzed the SPEC95 benchmarks [53] to determine the available ILP in these programs. They found the *limit* ILP to be in the range of 55-4003. This limit ILP was found by examining all dynamic instructions in the program. They also report that with a 10,000 entry instruction window the available ILP in these programs ranged from 18-287.

Superscalar processors detect independent instructions by examining small sections of the program every cycle and so they cannot detect such distant parallelism. In this section we quantify the maximum IPC that can be extracted from SPEC 2000 programs with issue window capacities that can be accommodated on-chip in the near future. We also examine IPC sensitivity to processor issue width.

3.1.1 Experimental Methodology

For these experiments we simulated out-of-order superscalar processor configurations of varying issue-widths and instruction window capacities. We used a modified version of the sim-alpha [20] for this purpose. Since our goal is to examine the sensitivity of IPC to issue width and instruction capacity alone these experiments simulated perfect branch prediction and a perfect memory system. The capacities of other structures such as the register file and the

Integer	Inst. Skipped	Floating Point	Inst. Skipped
164.gzip	33.20	171.swim	119.60
175.vpr	19.97	172.mgrid	38.09
176.gcc	5.07	173.applu	214.59
181.mcf	33.63	177.mesa	63.99
197.parser	90.73	178.galgel	17.23
252.eon	20.73	179.art	6.63
253.perlbmk	77.66	183.equake	19.34
256.bzip2	43.12	188.ammp	92.89
		189.lucas	126.07

Table 3.1: The number of dynamic instructions (in billions) skipped for the SPEC 2000 benchmarks before simulating 100 million instructions.

re-order buffer were unconstrained.

We simulated benchmarks from the SPEC2000 suite with *ref* input sets executing on an out-of-order superscalar machine. The benchmarks are compiled for the Alpha EV6 instruction set. A region of 100 million dynamic instructions was identified for each benchmark using a simulation utility (SimPoint) developed by Sherwood *et al.* [61]. SimPoint uses basic block execution frequency to identify regions of the program that closely represent overall program characteristics. We simulated the region of the programs identified by SimPoint in an out-of-order simulator. Table 3.1 shows the number of dynamic instructions skipped for each benchmark before simulation. All experiments simulated 100 million dynamic instructions after skipping the initial number of instructions in Table 3.1.

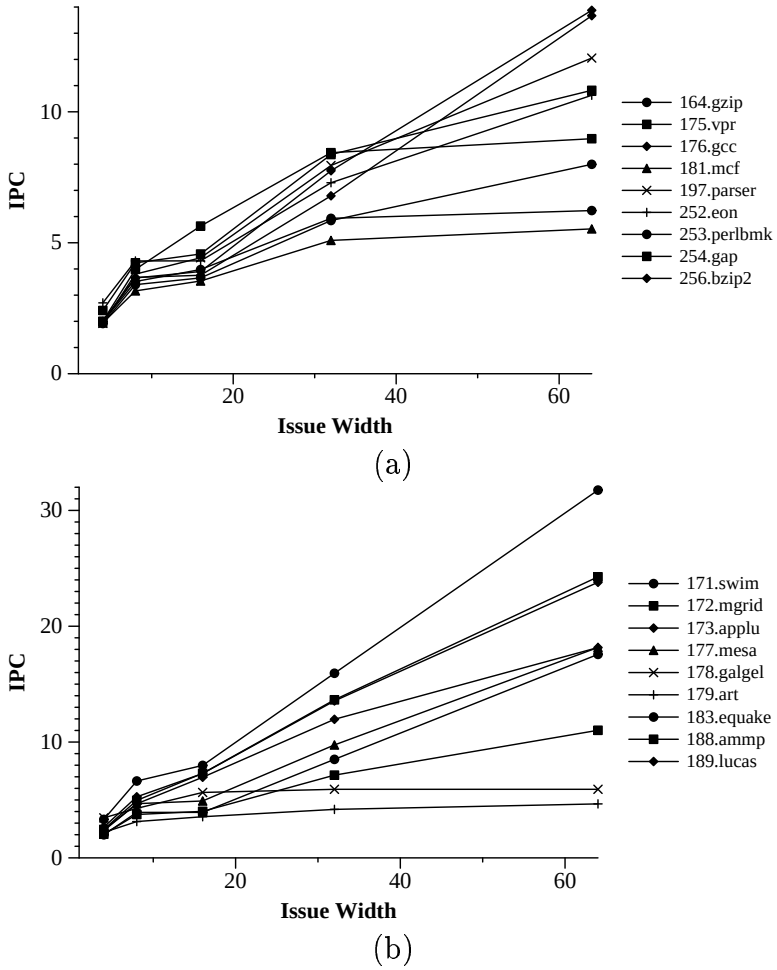

Figure 3.1: The IPC of SPEC 2000 benchmarks at different issue widths for a processor with a 2048 entry issue window.

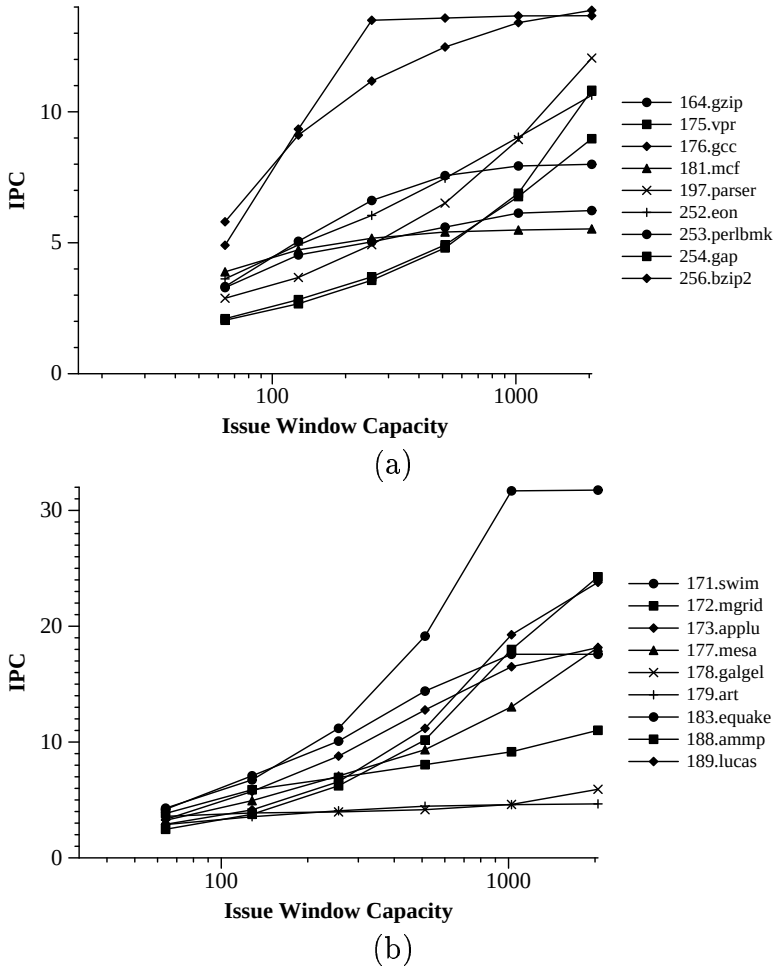

Figure 3.2: The IPC of SPEC benchmarks at different issue window capacities for a 64-wide processor

3.1.2 Results

Figure 3.1 shows a plot of IPC against issue width for a configuration with a 2048 entry issue window. The IPCs of both integer and floating point benchmarks increase as issue width is increased. For all benchmarks IPC increases when the processor issue width is increased. For some of the integer benchmarks such as 181.mcf, 253.perlbmk, and 254.gap IPC improvements plateau at an issue width of 32. However, for all other integer benchmarks IPC shows an improvement as the processor issue width is increased further. The IPCs of the integer benchmarks range between 5 to 14. Figure 3.1(b) shows the IPC of floating point benchmarks for processor configurations with increasing issue widths. For all the floating point benchmarks, with the exception of 179.art and 178.galgel, IPC increases as the processor issue width is increased. The IPCs of the floating point benchmarks range between 4 to 32.

Figure 3.2 shows the sensitivity of IPC to instruction issue window capacity, given a 64-wide processor. For some integer benchmarks, such as 181.mcf, 164.gzip, 253.perlbmk and 256.bzip2, IPC improvements plateau at a issue window capacity of 512 entries. For the other benchmarks IPC continues to improve as the issue window capacity is scaled. Among the floating point benchmarks 179.art and 178.galgel show very marginal IPC improvement as issue window capacities are increased. For two of the benchmarks—171.swim and 183.equake— IPC improvements plateau at an issue window capacity of 1024 entries. All other floating point benchmarks show an improvement in IPC as the issue window capacity is increased further.

We also examined the IPC that can be attained with a branch predictor similar to the Alpha 21264 and a real memory system comprising of two levels of cache. We observed that IPCs were in the range of 0.1-3.7 for the SPEC 2000 benchmarks. Instruction supply and data supply are fundamental problems to the design of any processor and innovations in these areas are required for improving IPC. There are several efforts directed at improving instruction supply [16, 28, 37, 46, 52, 57, 60, 75]. Similarly a significant effort is also being made to reduce memory access latency [4, 8, 17, 18, 38, 73]. However, this research addresses the issues in designing the execution core of the machine.

To achieve high performance, superscalar processors will have to issue more instructions every cycle, incorporate large on-chip structures with more access ports and still operate at high clock frequencies. However, increasing structure capacities and the number of ports also increases their access latency. Furthermore, as processes technology shrinks, the poor scaling of wires will further exacerbate the delay of on-chip structures. In a previous study, Agarwal *et al.* examined the effect of technology scaling on processor performance [2]. Their study shows that the performance of a conventional monolithic design will scale poorly with technology.

The twin goals of wider issue and high frequency are at odds with each other. Increasing issue width and structure capacities increases circuit complexity and area (e.g. greater number of register and cache ports) and so inhibits clock frequency. A natural solution to the problem of increasing

circuit complexity is to partition the architecture into clusters. Each of the processor's on-chip structures is divided among the clusters and therefore the complexity of each individual piece is reduced. In the rest of this chapter we provide a brief overview of previous work on partitioned architectures and describe in detail the baseline clustered processor that we use in later chapters.

3.2 Partitioned Architectures

Several architectures in the past have used resource partitioning to address the growing complexity of microprocessor design. The proposed designs range from partitioning just the execution core to multicomputer systems. In this section we present a short summary of previous work on partitioned architectures.

3.2.1 Very Long Instruction Word Processors

Clustering is widely used in very long instruction word (VLIW) processors. These machines issue a single long instruction every cycle. Each instruction has many tightly coupled independent operations that require a statically predictable number of cycles to operate. The ELI-512, proposed by Joseph Fisher [26], is an example of a VLIW processor. This processor has 16 clusters each with an ALU, a register file, and a memory bank. Instructions are statically scheduled by the compiler using a method called "trace scheduling".

The Multiflow family of machines is an example of a commercial im-

plementation of the VLIW architecture [45]. In this design also the physical registers are partitioned so that each cluster owns a subset of the registers. Functional units write values only into the local register file and to the remote register file via a shared global bus. If an instruction in one cluster requires an operand in another cluster the value has to be transferred over the global bus. The compiler generates explicit move instructions to perform such transfers.

3.2.2 Multithreaded Architectures

The HEP multithreaded processor proposed by Burton Smith [62] partitions the execution units, registers, and the data memory. The processing nodes in the processor are connected to banks of data memory via a switched network. Each processing element contains a 20-bit program counter (process status word) that sequences through a thread. An "access state" is associated with registers and data memory, to allow threads to synchronize.

3.2.3 Other Partitioned Architectures

Sohi *et al.* proposed the Multiscalar processor [27, 36, 63, 71] consisting of a collection of processing units. The processing units are organized as parallel pipelines and each unit consists of an instruction cache, functional units, and a register file. Every processing unit fetches, decodes, and executes instructions in parallel. In the Multiscalar execution model the program's control flow graph (CFG) is partitioned into tasks and each task is assigned to a processing unit. Multiple tasks execute in parallel on different processing

units.

RAW processors partition on-chip structures such as caches and register files among an array of processing "tiles" which are connected by a switched network [69, 72]. Instructions in a tile are executed in-order and the architecture relies on the compiler to schedule instructions onto the processing array. The TRIPS processor is another partitioned and scalable design proposed by Nagarajan *et al.* [47]. This design executes a statically mapped window of instructions on a substrate consisting of an array of ALUs. Though instructions are mapped statically onto the grid they execute dynamically in a data-flow fashion.

3.2.4 Clustered Superscalar Processors

Many modern implementations of superscalar processors partition the architecture into two clusters—an integer cluster and a floating point cluster. Since integer and floating-point are fundamentally different data types, placing them in different clusters partitions the design along a natural boundary. To operate at high frequencies future designs must incorporate multiple integer and floating point clusters. The Alpha 21264 is the first commercial superscalar processor to pursue such a clustered design [41]. This processor is a 7-stage, 4-wide machine. The execution stage of the integer pipeline is partitioned into two clusters with each cluster containing two integer units. The floating-point functional units are placed in a cluster of their own. Each integer cluster has a copy of the register file and when instructions complete

execution they write results into both register files. However, writing values to the register file of a *remote* cluster requires an additional clock cycle. The Alpha 21264 design partitions the register read, issue, and execute stages of the processor. Other stages such as renaming, instruction scheduling etc. are still unified and rely on monolithic structures.

Farkas *et al.* proposed a superscalar clustered architecture called the Multicluster architecture [22]. This design partitions the physical register file and the instruction issue window among the clusters. Functional units in one cluster can access the register file in that cluster alone. Special transfer instructions are used to transfer register values between clusters. However, unlike in clustered VLIW architectures these transfer instructions are generated dynamically by the hardware.

After instructions are fetched and decoded they are distributed to clusters based on the registers named by each instruction and the clusters to which the architectural registers have been assigned. Since instructions are assigned their registers at compile time the instruction cluster assignments are done statically (compile time). Instructions that are executed by a cluster are those that are issued by the instruction window of that cluster and after execution they write their outputs to the register file within the cluster.

The Multicluster architecture reduces the number of ports required for the register file and the instruction issue window. It also requires less complex instruction scheduling logic compared to a non-clustered design. In the Alpha 21264 processor the register file is replicated in all clusters. Since in-

struction outputs are written to all clusters, every register file requires enough write ports to sustain the entire processor's commit width. In contrast, the Multicluster processor partitions the register file among the clusters and every register file requires enough write ports to sustain the commit rate of just one cluster.

In the Multicluster architecture instructions are assigned to clusters at compile time. Others have proposed dynamic mechanisms to distribute instructions to clusters (i.e. *steering mechanisms*). Palacharla *et al.* proposed a *dependence* based steering method [50]. In this method, independent instructions are steered to the cluster with the lightest load. If an instruction consumes a value produced by another instruction then it is steered to the same cluster as its parent. If the source operands of an instruction are produced by parents in two different clusters then the instruction is assigned to the parent cluster with the lightest load. Canal *et al.* proposed the load-slice steering algorithm [14] which steers instructions that belong to the *backward slice* of a load to the same cluster. Baniasadi and Moshovos proposed the *mod3* steering method [5] in which first three instructions are steered to the first cluster, the next three instructions to the second cluster and so on. They also examined similar methods (i.e. scheduling 4 contiguous instructions to the same cluster and so on) and concluded that mod3 steering provides the best performance. Researchers have proposed several other dynamic steering policies. We describe these policies in Section 6.4.

3.2.5 Discussion

VLIW processors and the RAW machine rely on the compiler to statically schedule instructions. Compilers may schedule instructions using sophisticated optimization algorithms that are not feasible to build in hardware. In addition, the compiler can examine and perform optimizations across a larger number of instructions than dynamically scheduled processors. For certain classes of applications that have predictable control flow and regular data access patterns, such as scientific workloads, this type of architecture will perform better than dynamically scheduled superscalar processors. However, for other applications the compiler does not have access to run-time information such as memory access latencies, data access strides, and control path misprediction. So the static schedule produced by the compiler will not be as good as a dynamic instruction scheduling. For these applications statically scheduled processors will not perform better than a dynamically scheduled superscalar processor.

In the TRIPS processor, instructions are statically mapped onto the execution substrate but instruction issue is done dynamically. Using a combination of static placement and dynamic issue will enable such architectures to scale to wider issue widths than a superscalar processor. However, these architectures require significant changes to the instruction set and will break compatibility with older designs. In addition, they also rely on the compiler to generate large contiguous blocks of instructions (hyper-blocks). Currently there are research efforts directed at generating large hyper-blocks. The out-

come of these efforts will greatly influence the performance of the TRIPS architecture. Clustered superscalar processors use prediction mechanisms to determine the control flow within programs. They do not require an extensive compiler support as the TRIPS class of architectures. Furthermore, they are similar in design to monolithic superscalar architectures and maintain compatibility with older machines.

In this section we presented brief overview of partitioned architectures that have been proposed in literature. In the rest of this dissertation we examine the issues associated with the design of clustered superscalar processors. The baseline clustered processor that we study is similar to the Multicluster processor and is described in greater detail in the next section.

3.3 Baseline Clustered Architecture

Our baseline processor pipeline, shown in Figure 3.3, contains the following stages—fetch, decode, steer, rename, issue, register read, execute, and commit. The fetch, decode, steer, rename and commit stages are not partitioned. The issue, register read, and execute stages of the pipeline are partitioned into four clusters. The clusters are homogeneous and each cluster can execute all categories of instructions. Every cluster has two separate 128-entry instruction issue windows for integer and floating-point instructions. The physical registers are also partitioned across the clusters so that every cluster has two 128-entry physical register files, one for integer and one for floating-point instructions. The functional units in a cluster can access only

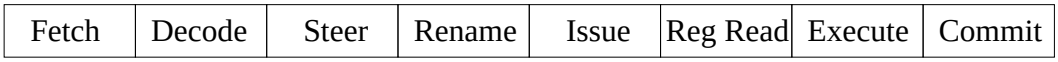

Figure 3.3: Pipeline diagram of the baseline processor

the register files in that cluster. The physical register capacities were chosen to match the capacity of the instruction issue windows, so that performance is not unduly limited due to insufficient rename buffers. Each cluster can issue up to two integer instructions and two floating-point instructions every cycle. Therefore the processor can issue up to 16 instructions (8 integer and 8 floating-point) every cycle. In our experiments we evaluate a total of three machine configurations—8-wide, 16-wide, and 32-wide. The 8-wide and 32-wide configurations are also 4-cluster machines. We vary the issue width of individual clusters to change the overall processor issue-width.

In addition to the issue windows and the register files, a 64 KB level-1 data cache is also partitioned across the clusters so that each cluster has a 16 KB cache bank. For design simplicity, we decided to statically map addresses to the cache banks and have them operate as one logical unit. A static mapping of addresses to cache banks also simplifies the partitioning of the Load/Store queue and makes it easier to detect memory dependencies than dynamic address mapping schemes. The level-1 cache banks are connected to a 2 MB, 12 cycle, direct mapped level-2 cache via a common bus.

Figure 3.3 shows the different operations that are performed along the

pipeline. Every cycle up to 16 instructions are fetched[1], and in subsequent stages these instructions are decoded. In the steer stage, every instruction is assigned a cluster on which it will be executed. There are several steering techniques that have been previously proposed [5, 14, 50]. We employ the steering methods described in Section 3.2.4 as our baseline.

After being assigned clusters the instructions move to the register rename stage where the output operand of every instruction is assigned a unique physical register. Instructions are assigned output registers from the cluster to which they have been steered. If a cluster does not have free physical registers available then the pipeline stalls until a physical register is released. After instructions are renamed, in the next cycle they are placed in the issue window of the cluster to which they have been assigned. Instructions wait in the issue window until both their source operands are ready after which they are considered for selection. *Selection logic* assigns ready instructions to specific functional units. Once selected, instructions progress to the register read stage where they read source operand values from the physical register file or the bypass latches. However, some instructions may require source operands that are produced in other clusters. These operands are read from another structure, called the *transfer buffer*. This method of inter-cluster communication was proposed by Farkas *et al.* for the Multicluster architecture [22]. We describe the role of transfer buffers for inter-cluster communication in detail later in

[1]For all configurations the fetch width matches the issue width. Thus, for the 8-wide processor configuration 8 instructions are fetched every cycle.

this section. After obtaining their operands, instructions move to the appropriate functional unit in the execute stage. The instructions are executed and the results are written into the physical register file. In the next cycle, the instruction proceeds to the commit stage where instructions are committed in order. Up to 16 instructions can be committed every cycle.

Figure 3.4 shows the issue and execute stages of the clustered processor in detail. For simplicity, we show only two clusters. An instruction (I1) that is assigned to one cluster (cluster 0) may require a source operand that is produced in another cluster (cluster 1). Such inter-cluster dependencies are detected in the rename stage of the pipeline and a special transfer instruction is inserted into the issue window of cluster 1. The transfer instruction will copy the required value from the register file in cluster 1 to a transfer buffer in cluster 0. The transfer instruction is selected for issue only if the required register value is ready and cluster 0 has a free transfer buffer entry available. The transfer instruction not only copies the required operand value to cluster 0 it also wakes-up instruction I1. If the other operand of I1 is ready then it can be selected for issue. Once selected I1 reads its *remote operand* from the transfer buffer and releases the corresponding entry of the transfer buffer. This clustering scheme is similar to the one proposed by Farkas *et al.* [22].

Unlike the Multicluster architecture, our baseline processor has a partitioned level-1 data cache. Therefore, some memory instructions that are steered to one cluster (cluster 0) may require access to cache banks in other clusters (cluster 1). Such a memory instruction will place a request in cluster

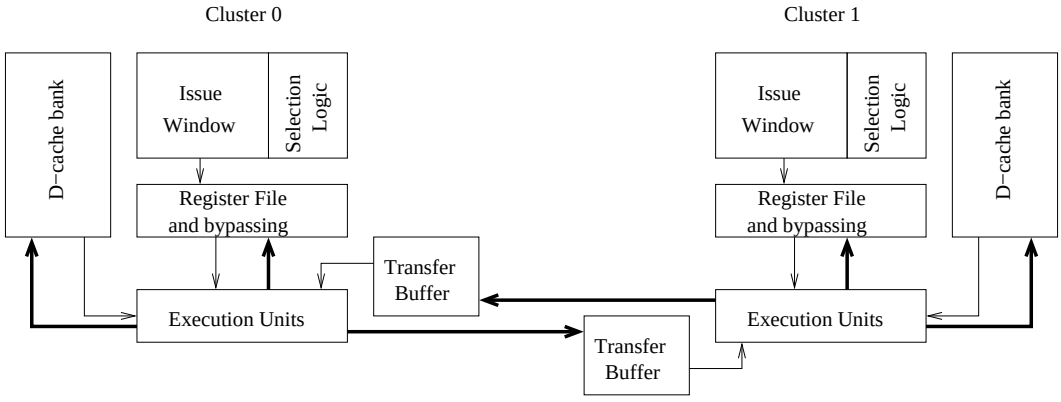

Figure 3.4: The issue, register read and execute stages of a clustered superscalar processor

1's *remote memory access queue*, a structure similar to a transfer buffer. This request will compete for cache ports with cluster 1's functional units. When a port becomes available the appropriate memory access is performed and a completed signal is sent to cluster 0. Requests in the remote memory queue are processed in FIFO order.

There are four clusters in our baseline processor and we envision these clusters laid out in the form of a square. Adjacent clusters have an inter-cluster communication latency of 1 cycle while clusters along the diagonal have an inter-cluster latency of 2 cycles. Therefore, once a transfer instruction has read a register value it will take up to 2 cycles for the value to be placed in the remote cluster's transfer buffer (1 cycle if the remote cluster is adjacent). Similarly, memory instructions that access remote cache banks have an additional inter-cluster communication latency. However, these instructions pay a round trip

penalty of up to 4 cycles (2 cycles if the clusters are adjacent). Note that the variable level-1 data cache access latency in our baseline processor will increase the complexity of the instruction scheduling logic.

In the Alpha-21264 processor there is a 1-cycle communication delay between the two integer clusters [41]. We assume a similar communication latency between adjacent clusters. At future technologies wire delays will scale poorly relative to transistor switching speeds and so inter-cluster communication delays are likely to increase. Therefore, the inter-cluster communication latencies that we assume are aggressive.

Our baseline model of a clustered processor divides the total on-chip resources among the clusters rather than replicate them like in the Alpha 21264. The primary reason we considered this design is because it is more scalable than an Alpha 21264-like clustered model. As processor widths increase, an Alpha 21264-like clustering mechanism will require a greater number of ports for register files and therefore increase their access latencies. Such a design will not scale well as issue width increases.

3.4 Summary

In this chapter we examined the amount of IPC that can be extracted from SPEC 2000 benchmarks. We found that with an issue window capacity of 2048 entries, the IPCs of these benchmarks can range from 4 to 32. The IPC of current processors typically sustain an IPC of less than 4. This suggests that significant performance improvements can be gained by extracting more

IPC from programs. However, control dependences in the program make it difficult to expose the available ILP to the processor. For future wide-issue designs to be viable significant effort has to be devoted to improving instruction supply. Several studies are directed towards addressing this problem [16, 28, 37, 46, 52, 57, 60, 75]. However, our research addresses the issues in designing the execution core of the machine.

To support wider issue the capacities of on-chip structures and the number of read and write ports must be increased. These structures must also have a low access latency and so they cannot be designed as monolithic units. Clustering is a natural solution to manage increasing on-chip complexity. In this chapter we provided a detailed description our baseline clustered processor. While clustering reduces the complexity of on-chip structures it introduces other bottlenecks and inefficiencies. The focus of our work is on reducing the bottlenecks from clustering. In Chapter 4 we evaluate the performance of our baseline clustered design and quantify the effect of these bottlenecks. In Chapters 5 and 6 we propose solutions to mitigate the effect of these bottlenecks.

Chapter 4

Bottlenecks in Clustered Architectures

Superscalar processors are moving towards clustered designs in order to increase clock frequency [33, 41]. Future wide-issue processors have to be partitioned to reduce the latency of critical on-chip structures such as register files and caches. Though clustered superscalar processors can be clocked at higher frequencies compared to un-clustered (monolithic) ones, the partitioned design introduces a few factors that reduce IPC. In this chapter we compare the performance of an ideal monolithic machine with configurations of a clustered processor employing three different steering policies. Our goal is to quantify the performance effect of the bottlenecks introduced by clustering.

4.1 Quantifying the Effect of Bottlenecks

For superscalar processors branch prediction and memory dependence prediction are two significant performance bottlenecks. Improvements in these areas is essential for wide-issue processors to be useful. Several other research efforts are directed at improving instruction supply [16, 28, 37, 46, 52, 57, 60, 75]. However, this research addresses the issues in designing the execution core of the machine. In our studies we evaluate configurations with perfect

branch prediction and with a tournament branch predictor like in the Alpha 21264 [41]. The perfect prediction studies were done to fully expose bottlenecks in the execution core which will otherwise be masked by other bottlenecks.

Our baseline clustered processor is similar to the Multicluster architecture and its organization is described in detail in Section 3.3. To model this clustered processor we used sim-alpha, a simulator developed by Desikan *et al.* [20]. We modified this simulator to support clustering and also the steering policies described in Section 3.2.4. We simulated benchmarks from the SPEC 2000 suite executing on a clustered machine, described in Section 3.3, and on a monolithic (one cluster) machine of equivalent issue width. In this section we quantify the IPC degradation that a clustered processor suffers in comparison with an *ideal* monolithic machine.

The monolithic machine is a one-cluster machine with the same amount of on-chip resources as the baseline clustered processor. This machine has two instruction issue windows—one for integer and one for floating point. The selection logic examines all instructions in the instruction window to select instructions to issue. Up to 8 integer and 8 floating point instructions can issued every cycle from the issue windows. Furthermore, up to eight result tags are broadcast to all entries in each issue window every cycle to wake up dependent instructions. The functional units in the monolithic machine read operands from centralized integer and floating point register files. Each register file has 16 read ports and 8 write ports and can be accessed in one cycle. Also, the monolithic machine has a 64KB unified data cache with a 3

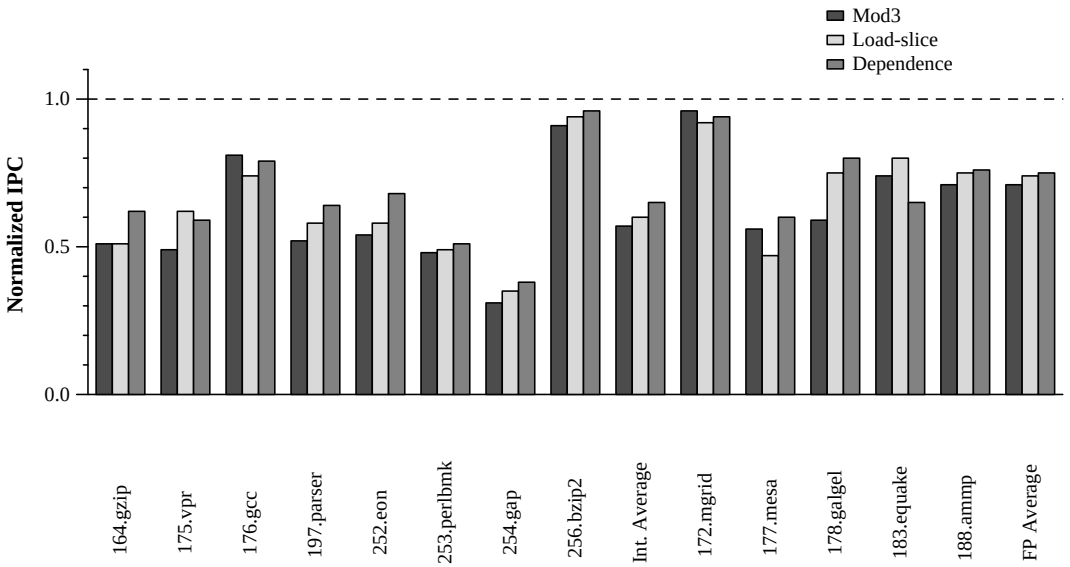

Figure 4.1: The IPC of a 16-wide 4-cluster processor normalized by the IPC of a monolithic processor. Both the clustered processor and the monolithic machine configurations simulated perfect branch prediction and perfect memory disambiguation.

cycle hit latency and 8 read-write ports to support the execution of up to eight memory instructions every cycle.

Large, centralized structures have long access latencies and therefore a monolithic machine, unlike clustered processors, cannot operate at high frequencies. Clustering is a technique that is used to partition microarchitectural structures and reduce their access latency. However, partitioning the architecture into clusters also reduces IPC compared to an ideal monolithic machine. In this section we quantify the IPC degradation by clustering.

Figure 4.1 shows the IPC of a 16-wide baseline clustered processor

normalized by the IPC of an ideal monolithic machine with identical resources. The absolute IPC numbers are listed in Appendix B. Few benchmarks—181.mcf, 171.swim, 189.lucas, 173.applu, and 179.art—have high miss rates in the level-1 and level-2 data caches. Changes in the microarchitecture (such as clustering) not not alter their IPC. Therefore, we do not show results for these benchmarks in our studies. For the clustered processor, we evaluated the performance of the three steering policies—mod3, load-slice and dependence. These policies are described in detail in Section 3.2.4. The IPC of the clustered processor is lower than that of the monolithic machine for all three steering policies. On average, the IPC of integer benchmarks is reduced by 39% and that of floating point benchmarks by 27% compared to a monolithic machine. Of the three steering policies mod3 steering shows the greatest degradation in IPC (37%), compared to an ideal monolithic machine, followed by load-slice steering (35%) and dependence steering (32%).

In addition to the baseline 16-wide processor, we examined the effect of clustering on 8-wide and 32-wide processors partitioned into four clusters. Table 4.1 shows the average reduction in the IPC of clustered machines compared to monolithic machines with equivalent resources. We observed that clustering causes significant IPC degradation for processor issue widths that are wider and narrower than our baseline. On average, for both 8-wide and 32-wide processors the dependence steering policy does better than mod3 and load-slice.

We also examined the IPC degradation due to clustering by simulating

Steering	8-wide		16-wide		32-wide	
	Int.	FP	Int.	FP	Int.	FP
mod3	53%	40%	43%	29%	51%	41%
load-slice	56%	46%	40%	26%	44%	33%
dependence	51%	44%	35%	25%	39%	30%

Table 4.1: Average reduction in the IPC of clustered processors compared to monolithic processors with equivalent resources. Both the clustered processor and the monolithic machine configurations simulated perfect branch prediction and perfect memory disambiguation.

configurations with an Alpha 21264-like branch predictor. These configurations used a store-wait predictor, similar to the Alpha 21264 [41], to predict dependences between load and store instructions. If the memory dependence of a load instruction is mis-predicted and it incorrectly executes ahead of a previous store instruction the pipeline is flushed and execution resumes from the offending load. For such a configuration we found the IPC of a clustered processor degraded by 46% for mod3, 35% for load-slice, and 34% for dependence steering.

Three factors degrade the IPC of clustered processors compared to a monolithic design. First is the communication delay to access remote operand values and remote memory values. The monolithic machine has a centralized register file and all functional units can access any entry in the register file in a single cycle. In a clustered processor functional units that are within one cluster have single cycle access to the register file in that cluster. Operand values that are stored in other clusters can be accessed only by incurring additional inter-cluster communication penalty. The second factor is poor utilization of

processor resources. In clustered processors on-chip resources such as physical registers, functional units, and issue windows are partitioned among clusters and this partitioning reduces overall efficiency. For example, in some cycles there will be more ready instructions available than the processor's issue width. We refer to these cycles as *issue-limited* cycles. Issue-limited cycles represent lost opportunity—situations where a wider-issue machine could have improved performance. A 16-wide monolithic machine can dispatch ready instructions to any of its functional units. But, in a clustered machine of similar width, instructions are directed to execute on specific clusters. In a given cycle, if one cluster is issue-limited, the extra ready instructions cannot be executed on other (lightly loaded) clusters. Therefore, a clustered design has potentially more issue-limited cycles compared to a monolithic design of similar issue width. Furthermore, in a clustered processor the instruction issue window is partitioned among the clusters. As described in Section 3.3, each cluster has a 128-entry instruction issue window. Instructions that are steered to a cluster in the Steer stage are eventually placed in that cluster's issue window. If the issue window is full the pipeline is stalled until an entry becomes available. Such structure capacity stalls also degrade performance.

The third factor that degrades the IPC of clustered machines is the extra transfer instructions used for inter-cluster communication. These instructions do not perform useful computation but they still consume processor resources such as entries in the instruction issue window and issue bandwidth. Thus they aggravate issue-limited cycles and structure capacity stalls.

To improve IPC we have to design wide-issue processors. These designs have to be partitioned into clusters so that they can operate at high clock frequencies. However, clustering introduces bottlenecks that cause significant IPC degradation compared to an ideal monolithic design. The goal of our research is to enable the design of partitioned wide-issue machines that operate at high frequency and also sustain instruction throughput comparable to a monolithic design.

4.2 Quantifying the Effect of Individual Bottlenecks

In this section we quantify the maximum IPC improvement that can be made over the baseline clustered processor if each bottleneck were individually and completely removed. Note that the configurations that we simulate for this study are not realistic designs and they were used only to examine the potential improvement that could be obtained by removing each individual bottleneck.

4.2.1 Transfer Instructions

To quantify the maximum IPC improvement that can be obtained by eliminating transfer instructions we simulated a clustered configuration where transfer instructions do not take up issue window slots or consume issue bandwidth. Figure 4.2 shows the IPC of the baseline clustered processor, with and without the transfer instruction bottleneck, normalized by the IPC of the monolithic configuration. The solid shaded bars on this graph are the IPC of

the baseline clustered processor normalized by the IPC of the monolithic machine. The patterned bars represent the IPC improvement that can be attained over the baseline if all transfer instructions were removed. These experiments simulated configurations with perfect branch prediction and perfect memory disambiguation. Most benchmarks, regardless of the steering policy, show an improvement in IPC over the baseline clustered processor when the transfer instruction bottleneck is removed. On average, the IPC of the integer benchmarks improved by 26% for mod3 steering, 2% for load-slice steering and 3% for dependence steering over their corresponding baselines. The floating-point benchmarks show an improvement of 25% for mod3 steering, 9% for load-slice steering and 6% for dependence steering. Furthermore, if transfer instructions are completely eliminated, the mod3 steering policy performs better than dependence and load-slice steering for all benchmarks except 178.galgel.

The mod3 steering policy attempts to balance the instruction workload evenly by steering an equal number of instructions to every cluster. However, this policy generates more transfer instructions than load-slice and dependence steering. Table 4.2 shows the number of transfer instructions executed by the baseline clustered processor as a fraction of the total number of instructions. On average, transfer instructions constitute 48% of the instructions executed by the clustered machine with mod3 steering. In contrast, transfer instructions constitute only 10% of the total instructions for load-slice steering and dependence steering. Both the load-slice and dependence steering policies favor reduced inter-cluster communication at the expense of higher workload

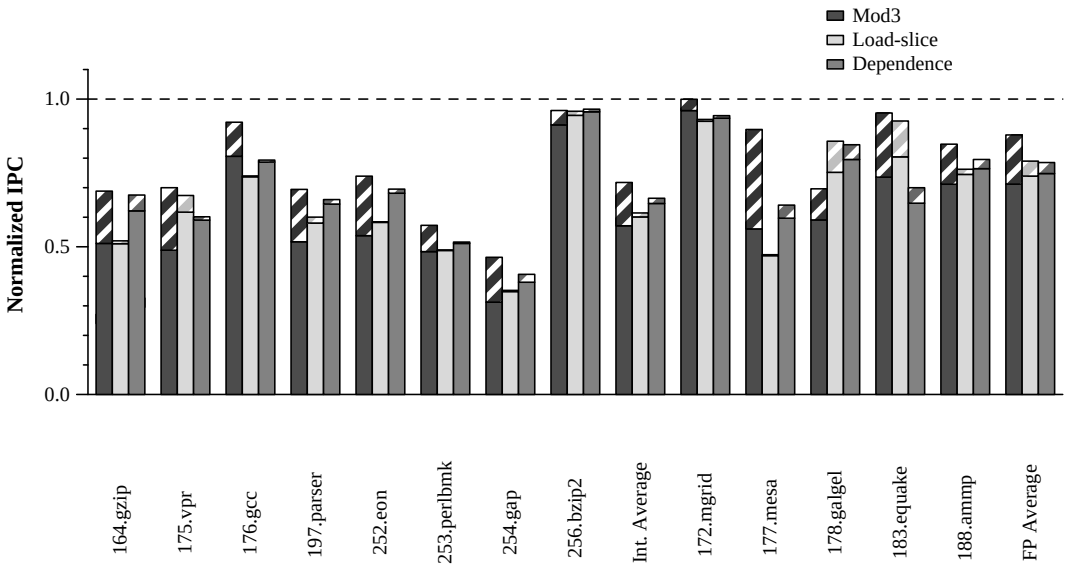

Figure 4.2: The IPC of a 16-wide clustered processor configuration, with and without the transfer instruction bottleneck, normalized by the IPC of the ideal monolithic machine. Both the clustered processor and the monolithic machine configurations simulated perfect branch prediction and perfect memory disambiguation.

imbalance and so they require fewer transfer instructions. Therefore, removing transfer instructions benefits the mod3 steering policy to a greater extent than load-slice and dependence steering policies.

Note that the mod3 steering policy has less state and is simpler to implement compared to load-slice or dependence steering. However, it generates a significant number of transfer instructions and could potentially consume more power than the other two policies.

We also simulated monolithic and clustered processor configurations with Alpha 21264-like branch and memory dependence predictors. Figure 4.3

Benchmark	Mod3	Load-slice	Dependence
164.gzip	49%	3%	13%
175.vpr	49%	23%	8%
176.gcc	43%	1%	5%
197.parser	46%	11%	8%
252.eon	49%	1%	5%
253.perlbmk	45%	2%	3%
254.gap	49%	5%	16%
256.bzip2	41%	10%	7%
172.mgrid	57%	16%	18%
177.mesa	50%	1%	11%
178.galgel	51%	18%	18%
183.equake	43%	19%	7%
188.ammp	47%	8%	10%
Average	48%	9%	10%

Table 4.2: Transfer instructions as a fraction of the total number of instructions executed by the baseline clustered processor.

shows the IPC of the baseline clustered processor, with and without the transfer instruction bottleneck, normalized by the IPC of the monolithic configuration. On average, the IPC of the integer benchmarks improved by 16% for mod3 steering, 5% for load-slice steering and 3% for dependence steering over their corresponding baselines. The floating-point benchmarks show an improvement of 13% for mod3 steering, 5% for load-slice steering and 6% for dependence steering.

The configurations that simulate perfect branch prediction, discussed in Figure 4.2, show a greater improvement when transfer instructions are removed than the configurations with the Alpha 21264-like branch prediction. For these configurations the pipeline is always full of useful instructions and

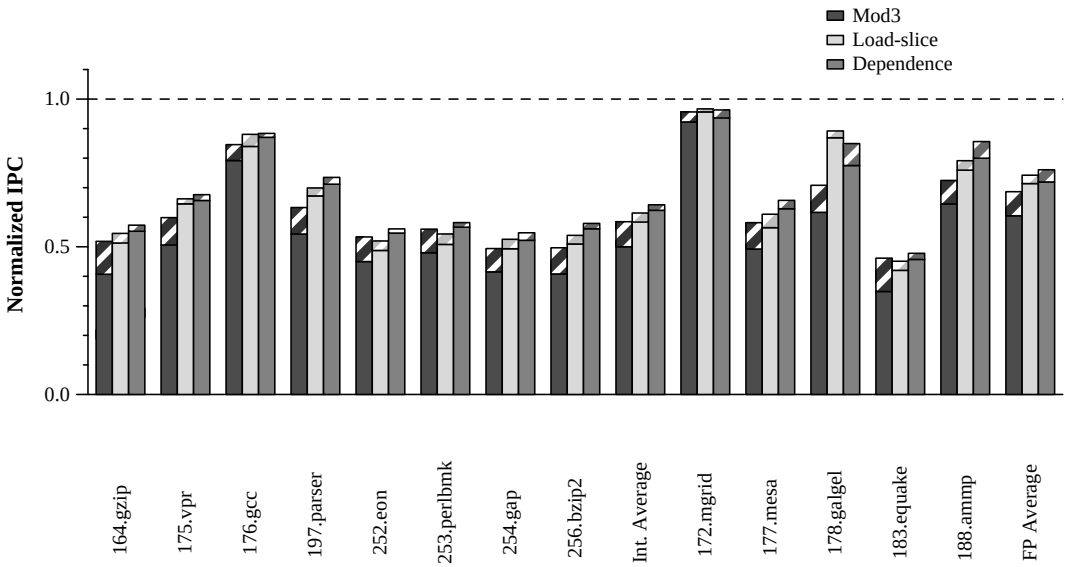

Figure 4.3: The IPC of a 16-wide clustered processor configuration, with and without the transfer instruction bottleneck, normalized by the IPC of the ideal monolithic machine. Both the clustered processor and the monolithic machine configurations used Alpha 21264-like branch prediction and memory dependence prediction.

every transfer instruction that is generated consumes resources, such as instruction issue window entries and issue bandwidth, and denies it to useful instructions (i.e. instructions that are actually performing computation). In the configurations that simulate Alpha 21264-like branch prediction the pipeline has both useful (correct path) and mis-predicted instructions. In this case, some transfer instructions consume resources that would have otherwise been allocated to mis-predicted instructions. Such transfer instructions do not affect the IPC of the machine and therefore the effect of removing the transfer instruction bottleneck appears less significant. However, as branch predic-

Steering	8-wide		16-wide		32-wide	
	Int.	FP	Int.	FP	Int.	FP
mod3	54%	52%	26%	25%	29%	29%
load-slice	5%	13%	2%	9%	2%	6%
dependence	8%	10%	3%	6%	1%	3%

Table 4.3: Average improvement in the IPC of clustered processors when the transfer instruction overhead is removed. Both the clustered processor and the monolithic machine configurations simulated perfect branch prediction and perfect memory disambiguation.

tion and memory dependence prediction techniques improve in the future, the transfer instruction bottleneck will become more severe.

In addition to a 16-wide processor we also examined the effect of removing the transfer instruction bottleneck for 8-wide and 32-wide processors. All the three clustered processors that we evaluated are 4-cluster machines. Table 4.3 shows the average improvement in IPCs when the transfer instruction bottleneck is removed. The effect of the transfer instruction bottleneck reduces as the issue width of the processor is increased. The smaller issue width processors have a lower issue bandwidth and therefore removing the transfer instruction bottle improves their IPC to a greater extent.

4.2.2 Inter-cluster Communication Delay

As described in Section 3.3, our baseline clustered processor partitions the register file and the level-1 data cache. Though the data cache is physically partitioned it operates as one logical unit. For design simplicity, we statically map addresses to cache banks. Therefore, a given cache line in memory will

always map to the same cache bank. Each cluster has direct access to one cache bank and remote banks are accessed by placing requests in the appropriate *remote request queue*. Since the data cache is partitioned among the clusters some memory instructions that are steered to one cluster (cluster 0) may require to access cache banks in other clusters (e.g. cluster 1). Such memory instructions will gain access to the cache bank in cluster 1 by queuing their request in cluster 1's *remote request queue*. When a port becomes available the appropriate memory access is performed and the value is forwarded to cluster 0. Effectively, clusters have fast access to memory addresses that map to the local cache bank but pay additional communication penalty if the address maps to remote cache banks. This additional communication penalty is equal to twice the communication latency (round-trip time) between the clusters involved.

Since the register file is also partitioned, some instructions that are steered to cluster 0 may require input register operands that are in other clusters (e.g. cluster 1). Such inter-cluster dependences are detected in the steer stage and a transfer instruction is inserted into cluster 1 to transfer the appropriate value. The communication penalty to transfer operand values is equal to the communication delay between the clusters involved in the transaction.

To quantify the maximum IPC improvement that can be obtained by eliminating inter-cluster communication we simulated a clustered configuration with no inter-cluster communication cost (0 cycles). Figure 4.4 shows the IPC of the baseline clustered processor, with and without the inter-cluster

communication bottleneck, normalized by the IPC of the ideal monolithic machine. These experiments simulated configurations with perfect branch prediction and perfect memory disambiguation. The solid shaded bars on this graph are the IPC of the baseline clustered processor normalized by the IPC of the monolithic machine. The patterned bars represent the IPC improvement that can be attained over the baseline if inter-cluster communication were 0 cycles. On average, the IPC of the integer benchmarks improved by 29% for mod3 steering, 10% for load-slice steering and 12% for dependence steering over their corresponding baselines. The floating-point benchmarks show an improvement of 23% for mod3 steering, 8% for load-slice steering and 5% for dependence steering. Note that, if the inter-cluster communication bottleneck is completely eliminated, the mod3 steering policy performs better than dependence and load-slice steering for the majority of benchmarks.

The mod3 steering policy attempts to balance the instruction workload evenly by steering an equal number of instructions to every cluster. Though this policy makes the best use of the processor's issue window slots it does so at the cost of increasing the inter-cluster communication. Table 4.4 shows the number of remote operands required by each steering policy as a fraction of the total number of operands read during execution. On average, remote operands constitute 66% of all operands read by the clustered machine with mod3 steering. In contrast, remote operands constitute only 8% of the total operands read for load-slice and dependence steering. Therefore, removing the inter-cluster communication bottleneck benefits the mod3 steering policy to a

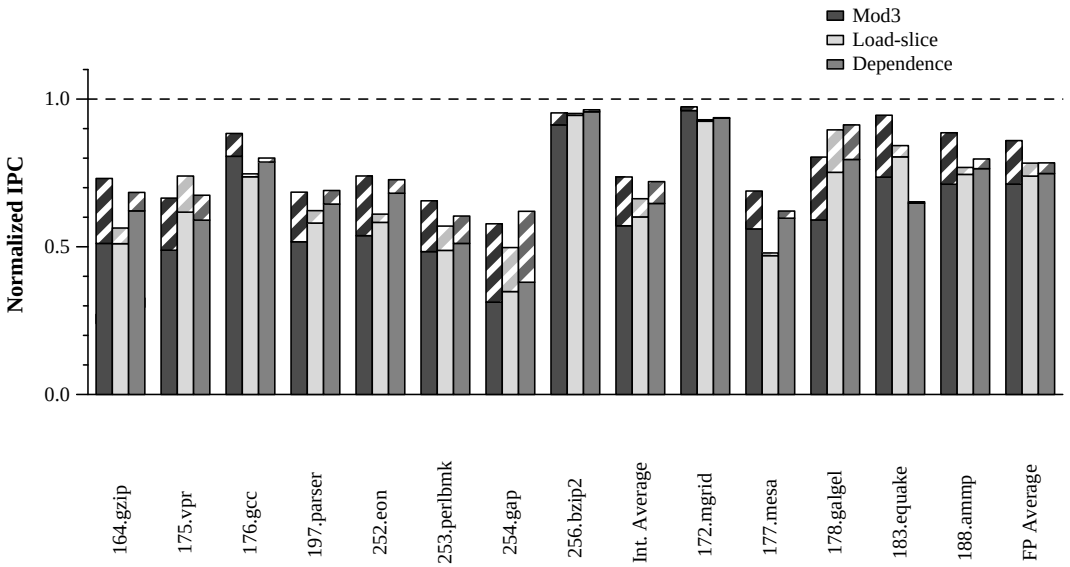

Figure 4.4: The IPC of a 16-wide clustered processor configuration, with and without the inter-cluster communication bottleneck, normalized by the IPC of the ideal monolithic machine. Both the clustered processor and the monolithic machine configurations simulated perfect branch prediction and perfect memory disambiguation.

greater extent than load-slice and dependence steering policies.

For load-slice and dependence steering policies the inter-cluster communication and the transfer instruction bottlenecks degrade IPC by approximately the same extant. In the case of mod3 steering, for benchmarks 164.gzip, 176.gcc, 253.perlbmk, 254.gap, 178.galgel, and 188.ammp the inter-cluster communication bottleneck is more significant than the transfer instruction bottleneck (Figure 4.2). While for benchmarks 175.vpr and 177.mesa the transfer instruction bottleneck is more significant. For the other benchmarks, both bottlenecks degrade IPC by approximately the same extant.

Benchmark	Mod3	Load-slice	Dependence
164.gzip	68%	2%	10%
175.vpr	66%	21%	6%
176.gcc	55%	1%	4%
197.parser	62%	9%	7%
252.eon	69%	1%	4%
253.perlbmk	64%	1%	3%
254.gap	68%	3%	13%
256.bzip2	56%	8%	6%
172.mgrid	80%	12%	14%
177.mesa	71%	1%	9%
178.galgel	78%	16%	16%
183.equake	60%	18%	6%
188.ammp	66%	7%	8%
Average	66%	8%	8%

Table 4.4: Remote operand accesses as a fraction of the total number of operands read during execution by the baseline clustered processor.

We also simulated monolithic and clustered processor configurations with Alpha 21264-like branch and memory dependence predictors. Figure 4.5 shows the IPC of the baseline clustered processor, with and without the inter-cluster communication bottleneck, normalized by the IPC of the monolithic configuration. On average, the IPC of the integer benchmarks improved by 64% for mod3 steering, 41% for load-slice steering and 35% for dependence steering over their corresponding baselines. The floating-point benchmarks show an improvement of 41% for mod3 steering, 24% for load-slice steering and 25% for dependence steering. The configurations that simulate perfect branch prediction, discussed in Figure 4.4, show lower improvement when the inter-cluster communication bottleneck is removed as compared to the configu-

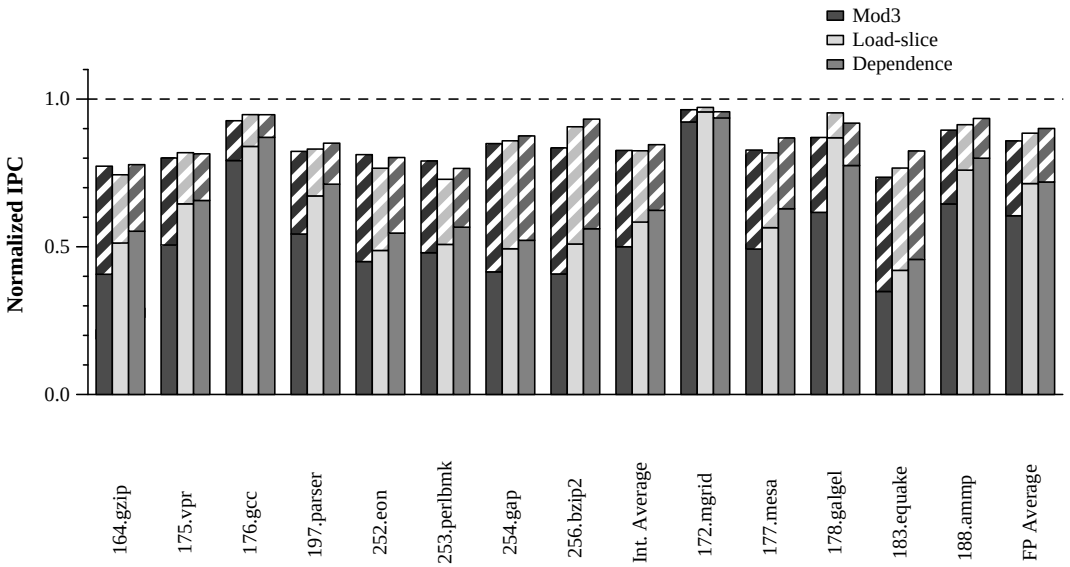

Figure 4.5: The IPC of a 16-wide clustered processor configuration, with and without the inter-cluster communication bottleneck, normalized by the IPC of the ideal monolithic machine. Both the clustered processor and the monolithic machine configurations used Alpha 21264-like branch prediction and memory dependence prediction.

rations with the Alpha 21264-like branch predictor. For the perfect prediction experiments the pipeline is frequently full of useful instructions and at a given cycle there are more ready instructions available than in the case with the Alpha 21264-like predictor. Therefore, in the perfect prediction experiments, a larger portion of the time spent waiting for remote operands is overlapped by other executing instructions. For this reason, removing the inter-cluster communication bottleneck shows a greater IPC benefit in the experiments with the Alpha 21264-like predictor.

Note that for all configurations in which we simulate Alpha 21264-like

Steering	8-wide		16-wide		32-wide	
	Int.	FP	Int.	FP	Int.	FP
mod3	23%	12%	29%	23%	50%	39%
load-slice	4%	1%	10%	8%	26%	13%
dependence	8%	1%	12%	5%	23%	11%

Table 4.5: Average improvement in the IPC of clustered processors when inter-cluster communication latency is removed. For these experiments, all configurations simulated perfect branch prediction and perfect memory disambiguation.

branch prediction the IPC benefits from removing the inter-cluster communication bottleneck is greater than the benefits from removing the transfer instruction bottleneck (Figure 4.3). However, as branch prediction techniques improve, the IPC degradation due to inter-cluster communication reduces while the degradation due to transfer instructions will become more significant.

In addition to a 16-wide processor we also examined the effect of removing the transfer instruction bottleneck for 8-wide and 32-wide processors. All the three clustered processors that we evaluated are 4-cluster machines. Table 4.5 shows the average improvement in IPCs when the inter-cluster communication bottleneck is removed. The other two bottlenecks—transfer instructions and cluster resource limitations—are more significant in smaller issue-width processors. These bottlenecks mask the inter-cluster communication bottleneck. Therefore, IPC degradation due to the inter-cluster communication bottleneck is more significant in wide-issue processors than in small issue-width processors.

4.2.3 Cluster Resource Limitations

To quantify the effect of cluster resource limitations we simulated a clustered processor configuration where an individual cluster's resources are not restricted. For example, in an 16-wide configuration, each cluster can issue up to 16 instructions but the total number of instructions issued across all clusters in one cycle is still restricted to 16. Similarly, instructions are allowed to map to any issue window as long as the total number of instructions in all issue windows is less than 512. Figure 4.6 shows the IPC of the baseline clustered processor, with and without the cluster resource limitation bottleneck, normalized by the IPC of the monolithic configuration. The solid shaded bars on this graph are the IPC of the baseline clustered processor normalized by the IPC of the monolithic machine. The patterned bars represent the IPC improvement that can be attained over the baseline if the cluster resource limitation bottleneck is removed. These experiments simulated configurations with perfect branch prediction and perfect memory disambiguation. On average, the IPC of the integer benchmarks improved by 23% for mod3 steering, 53% for load-slice steering and 43% for dependence steering over their corresponding baselines. The floating-point benchmarks show an improvement of 3% for mod3 steering, 25% for load-slice steering and 24% for dependence steering.

The dependence steering mechanism works on the premise that dependent instructions could potentially be part of the critical path of the program. To minimize operand communication latency between producer and consumer

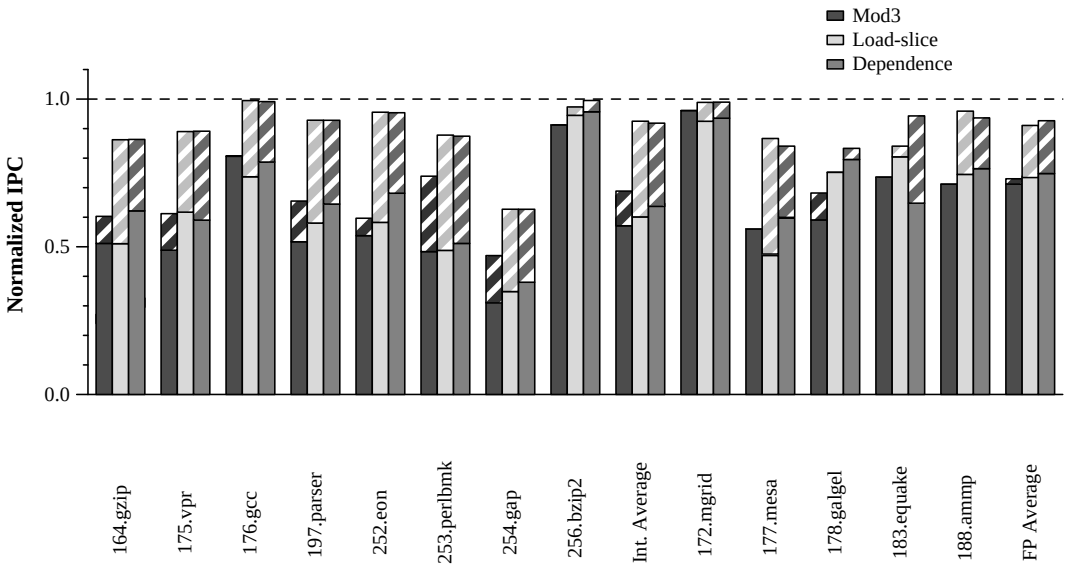

Figure 4.6: The IPC of a 16-wide clustered processor configuration, with and without the cluster resource limitation bottleneck, normalized by the IPC of the ideal monolithic machine. Both the clustered processor and the monolithic machine configurations simulated perfect branch prediction and perfect memory disambiguation.

instructions this policy attempts to steer dependent instructions to the same cluster. The load-slice steering policy steers all instructions that are part of the address computation for a load instruction to the same cluster. However, both these policies inundate a few clusters with instructions while starving others and thereby underutilize processor resources. We evaluated the workload imbalance in the following manner. Every cycle we rank order the clusters based on the number of instructions in each cluster (cluster with most instructions has highest rank). We track the number of instructions issued from every rank. Table 4.6 shows the fraction of instructions that are executed by each rank for

Benchmark	Rank 0	Rank 1	Rank 2	Rank 3
164.gzip	0.10	0.14	0.15	0.61
175.vpr	0.05	0.13	0.19	0.64
176.gcc	0.05	0.25	0.30	0.40
197.parser	0.14	0.17	0.19	0.49
252.eon	0.08	0.15	0.36	0.41
253.perlbmk	0.20	0.05	0.42	0.32
254.gap	0.13	0.14	0.40	0.36
256.bzip2	0.24	0.25	0.24	0.27
172.mgrid	0.19	0.25	0.27	0.30
177.mesa	0.10	0.15	0.30	0.46
178.galgel	0.20	0.23	0.27	0.30
183.equake	0.09	0.14	0.39	0.39
188.ammp	0.07	0.14	0.13	0.65

Table 4.6: Workload imbalance in a 16-wide, 4-cluster machine using dependence steering.

a 16-wide, 4-cluster machine with dependence steering. For most benchmarks dependence steering produces a skewed instruction distribution that results in under utilization of processor resources. We found that the load-slice steering policy also produces similarly skewed instruction distribution. The mod3 policy, on the other hand, distributes an equal number of instructions to every cluster and therefore makes better use of processor resources.

We collected statistics on the number of structure capacity stalls (i.e. the number of processor stalls because of a full issue window) and the number of instructions that were affected by limited per-cluster issue width. If in a given cycle a cluster has 4 ready instructions but can issue only two of them then we count the other two instructions as issue-limited instructions.

	Mod3		Load-slice		Dependence	
Benchmark	ILI	SCS	ILI	SCS	ILI	SCS
164.gzip	6.7	2.2	18.2	23.4	14.8	22.1
175.vpr	10.7	2.1	10.8	18.6	7.1	18.2
176.gcc	2.9	1.8	3.7	3.2	3.4	2.6
197.parser	5.3	1.6	8.1	17.9	6.2	14.0
252.eon	7.4	1.5	17.6	26.7	10.1	20.5
253.perlbmk	12.4	1.3	19.3	22.6	17.2	23.1
254.gap	1.9	0.7	9.3	12.0	10.7	13.4
256.bzip2	5.3	1.1	6.8	2.7	7.5	2.1
172.mgrid	1.1	1.4	2.0	3.6	1.7	2.0
177.mesa	4.7	1.2	9.3	18.4	6.4	11.2
178.galgel	1.1	1.0	2.3	1.7	1.6	1.1
183.equake	10.2	1.3	18.4	24.5	13.4	23.2
188.ammp	4.2	1.1	2.7	15.1	3.1	14.0
Average	5.7	1.4	9.9	14.6	7.9	12.9

Table 4.7: The number of issue-limited instructions (ILI) and structure capacity stalls (SCS) per 100 instructions. These statistics were collected for configurations with Alpha 21264-like branch prediction and memory dependence prediction.

Table 4.7 shows the number of issue-limited instructions and structure capacity stalls per 100 instructions. The load-slice and dependence steering produce a skewed instruction distribution. They steer the bulk of the instructions to just one or two clusters, in effect using few of the total number of available issue-window slots. Therefore, they suffer a large number of issue-limited instructions and structure capacity stalls. On the other hand, mod3 steering distributes an equal number of instructions to every cluster and so does not have as many issue-limited instructions and structure capacity stalls. For these reasons, removing the cluster resource limitation bottleneck improves

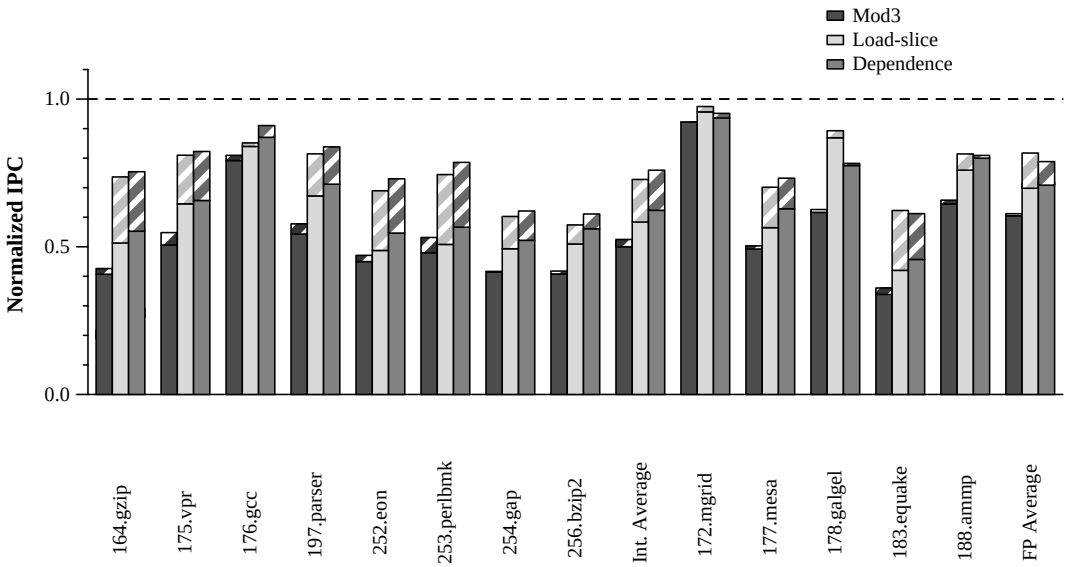

Figure 4.7: The IPC of a 16-wide clustered processor configuration, with and without the cluster resource limitation bottleneck, normalized by the IPC of the ideal monolithic machine. Both the clustered processor and the monolithic machine configurations used Alpha 21264-like branch prediction and memory dependence prediction.

the performance of load-slice and dependence steering to a greater extent than mod3 steering.

We also simulated monolithic and clustered processor configurations with Alpha 21264-like branch and memory dependence predictors. Figure 4.7 shows the IPC of the baseline clustered processor, with and without the cluster resource limitation bottleneck, normalized by the IPC of the monolithic configuration. On average, the IPC of the integer benchmarks improved by 5% for mod3 steering, 24% for load-slice steering and 23% for dependence steering over their corresponding baselines. The floating-point benchmarks

Steering	8-wide		16-wide		32-wide	
	Int.	FP	Int.	FP	Int.	FP
mod3	45%	22%	23%	3%	25%	10%
load-slice	113%	118%	53%	25%	50%	23%
dependence	90%	108%	43%	24%	42%	29%

Table 4.8: Average improvement in the IPC of clustered processors when cluster resource limitations are removed. For these experiments, all configurations simulated perfect branch prediction and perfect memory disambiguation.

show an improvement of 2% for mod3 steering, 12% for load-slice steering and 8% for dependence steering. The configurations that simulate perfect branch prediction, discussed in Figure 4.6, show a greater improvement when cluster resource limitations are removed than the configurations with the Alpha 21264-like branch prediction. For these configurations the pipeline is frequently full of useful instructions and at every issue-limited cycle useful instructions are prevented from being executed. In the configurations that simulate Alpha 21264-like branch prediction the pipeline has both useful (correct path) and mis-predicted instructions. In this case, some of the issue-limited cycles and structure capacity stalls are due to instructions along mis-speculated paths. Such stalls, caused by instructions on mis-speculated paths, do not affect overall performance and therefore the effect of removing the cluster resource limitation bottleneck is less significant.

In addition to a 16-wide processor we also examined the effect of removing the transfer instruction bottleneck for 8-wide and 32-wide processors. All the three clustered processors that we evaluated are 4-cluster machines.

Table 4.8 shows the average improvement in IPCs when the cluster resource limitation bottleneck is removed. The cluster resource limitation bottleneck degrades the IPC of small issue-width processors to a greater extent than wide-issue processors. Each cluster is an 8-wide processor can issue only two instructions (one integer and one floating point) every cycle. Since most benchmarks have an IPC greater than one, removing cluster resource limitations improves the IPC of this processor configuration significantly. However, as the total issue width of the processor is increased the issue bandwidth of every cluster increases and therefore the per-cluster resource limitation bottleneck becomes less significant.

4.3 Summary

In this chapter we quantified the IPC degradation of the baseline clustered processor compared to an ideal monolithic machine. Our evaluation shows that clustering degrades the IPC by 29-43% compared to a monolithic machine. Three bottlenecks reduce the IPC of the clustered processor–(1) inter-cluster communication delays, (2) transfer instruction overhead, and (3) cluster resource limitation. We quantified the improvement that can be obtained over the baseline processor if each bottleneck were individually removed.

We simulated monolithic and clustered processor configurations with Alpha 21264-like branch and memory dependence predictors. Table 4.9 shows the average improvement that can be attained over the baseline if each bottle-

Steering	Transfer Instruction		Inter-cluster Communication		Cluster Resource Limitation	
	Int.	FP	Int.	FP	Int.	FP
mod3	16%	13%	64%	41%	5%	2%
load-slice	5%	5%	41%	24%	24%	12%
dependence	3%	6%	35%	25%	23%	8%

Table 4.9: Average improvement in the IPC of a 16-wide 4-cluster processor when clustering bottlenecks are removed. Both the clustered processor and the monolithic machine configurations used Alpha 21264-like branch prediction and memory dependence prediction.

neck were individually removed. We found that the effects of the bottlenecks vary depending on the steering policy, the issue width of each cluster and the benchmark. All the three steering policies that we evaluated are affected most severely by the inter-cluster communication bottleneck followed by the cluster resource limitations and transfer instructions bottleneck. The mod3 steering policy is affected more severely by the transfer instruction overhead and by the inter-cluster communication bottleneck than the other policies. The cluster resource limitation bottleneck affects the load-slice and dependence policies to a greater extant than mod3 steering.

To fully expose the effect of the clustering bottlenecks we also simulated configurations with perfect branch prediction and memory disambiguation. Table 4.10 shows the average improvement that can be attained over the baseline if each bottleneck were individually removed. For these configurations we observed that the effect of transfer instruction overhead and the cluster resource bottleneck become more significant while the inter-cluster

Steering	Transfer Instruction		Inter-cluster Communication		Cluster Resource Limitation	
	Int.	FP	Int.	FP	Int.	FP
mod3	26%	25%	29%	23%	23%	3%
load-slice	2%	9%	10%	8%	53%	25%
dependence	3%	6%	12%	5%	43%	24%

Table 4.10: Average improvement in the IPC of a 16-wide 4-cluster processor when clustering bottlenecks are removed. For these experiments, all configurations simulated perfect branch prediction and perfect memory disambiguation.

communication bottleneck become less significant.

To discover and exploit parallelism in the instruction stream future superscalar processors will require larger on-chip structures and issue a greater number of instructions every cycle compared to current designs. These changes will increase the circuit complexity of these structures and therefore limit the achievable clock frequency. A natural solution to address the increase in complexity is to partition the various on-chip structures and organize them as clusters. However, our evaluation of the baseline clustered processor shows that partitioning a monolithic design significantly degrades instruction throughput. Clustered processors suffer from bottlenecks that degrade IPC. In the remainder of this dissertation we propose and evaluate techniques to address these bottlenecks. Our goal is to improve the IPC of a clustered processor to be closer to that of the ideal monolithic machine.

In the remainder of this dissertation we propose and evaluate methods to reduce the effect of the clustering bottlenecks. In Chapter 5 we propose

and evaluate techniques to remove the transfer instruction bottleneck by replacing them with hardware mechanisms. We propose dynamic instruction steering mechanisms in Chapter 6. These steering policies attempt to reduce inter-cluster communication latency and the bottlenecks from poor processor resource utilization.

Chapter 5

Reducing Transfer Instructions

Transfer instructions can be a significant overhead for clustered processors. Our experiments in Section 4.2.1 show that removing all transfer instructions can improve performance between 2-30%. In addition, if all transfer instructions are removed, a simple steering algorithm such as mod3 outperforms more sophisticated methods like dependence steering. In this chapter we examine three mechanisms to reduce this overhead. In the first mechanism, called register caching, we attempt to reduce the number of transfer instructions by caching remote register values. This mechanism eliminates multiple transfer instructions from being generated for re-used operands. The other two mechanisms—consumer-requested forwarding and hot-register based forwarding—replace transfer instructions with hardware signals.

5.1 Register Caching

In the baseline clustered processor a transfer instruction is generated for every inter-cluster dependence. As an example, for the code shown in Figure 5.1, instructions I3 and I4 both require register R24 as one of their input operands. Instruction I1, which produces R24, is assigned to cluster 0

```
I1: Add R24, R32, R47     (R24 <-- R32 + R47)       Cluster 0
I2: And R56, R52, R47     (R56 <-- R47 && R52)      Cluster 0
I3: Add R127, R24, R125 (R127 <-- R24 + R125)       Cluster 1
I4: Sub R128, R24, R126 (R128 <-- R24 - R126)       Cluster 1
```

Figure 5.1: Example of a stream of instructions with an inter-cluster dependence. Instructions 1 and 2 are assigned to cluster 0 while instructions 3 and 4 are assigned to cluster 1.

while I3 and I4 are assigned to cluster 1. In the situation described above, both I3 and I4 generate transfer instructions to transfer the value of R24 from cluster 0 to cluster 1. In this section we propose to augment each cluster with a register file cache that will be used to cache remote register values. In the register cache augmented processor only the first consumer instruction (I3) will generate a transfer instruction. The transfer instruction will copy the value of R24 from cluster 0 and place in the register cache in cluster 1. Instruction I4 will read its remote operand, register R24, from the register cache.

For the discussion in this section we use the following terminology. The first time that a register value is read after being produced is called "first use". All subsequent accesses to that operand are termed "re-uses". Thus, in the example shown in Figure 5.1, the first use of R24 is initiated by instruction I3 while instruction I4 re-uses R24. Table 5.1 shows the number of input operands that are re-used as a fraction of the total number of input operands. The table also shows how many times the operands were re-used. These statistics were generated by examining program traces in a functional simulator and are independent of the underlying microarchitecture. Approximately 45% of

Benchmark	All Re-used operands	Reused			
		1 time	2 times	3 times	>3 times
164.gzip	42%	23%	7%	5%	6%
175.vpr	43%	20%	8%	5%	10%
176.gcc	55%	20%	18%	9%	8%
197.parser	42%	21%	7%	3%	9%
252.eon	52%	23%	10%	6%	12%
253.perlbmk	46%	24%	9%	5%	8%
254.gap	41%	16%	12%	2%	12%
256.bzip2	36%	22%	8%	4%	2%
Average	45%	21%	10%	5	8%
172.mgrid	43%	16%	12%	6%	9%
177.mesa	46%	16%	10%	3%	16%
178.galgel	42%	23%	5%	6%	8%
183.equake	41%	20%	8%	5%	8%
188.ammp	42%	22%	9%	5%	6%
Average	43%	19%	9%	5%	9%

Table 5.1: The number of source operands that are re-used as a fraction of the total number of source operands.

all input operands read in integer benchmarks are re-uses. Similarly, 43% of input operands read in floating point benchmarks are re-uses. These statistics suggest that caching register values could improve performance by eliminating the transfer instructions that are generated for re-used operands.

We simulated a configuration of the baseline clustered processor augmented by register caches. The register cache in each cluster is large enough to hold all remote operands (384 remote registers). These experiments represent a best case situation wherein register caches are not capacity limited. They provide an upper bound on the performance benefits that can be obtained by

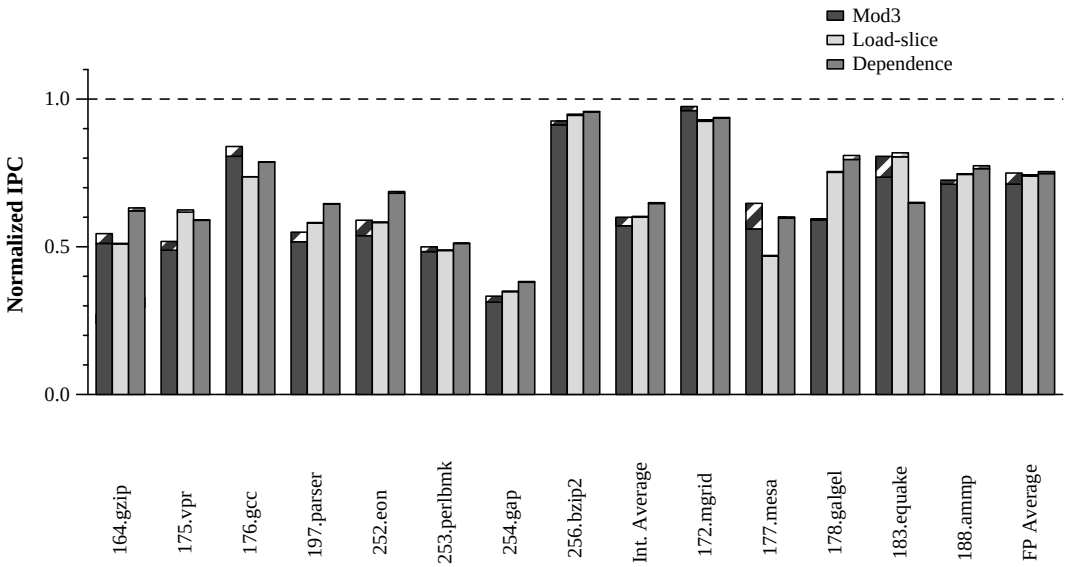

Figure 5.2: The IPC of a 16-wide 4-cluster processor, with and without register caching, normalized by the IPC of the ideal monolithic machine. Both the clustered processor and the monolithic machine configurations simulated perfect branch prediction and perfect memory disambiguation.

caching remote register data. Figure 5.2 shows the IPC of the baseline clustered processor, with and without register caching, normalized by the IPC of the monolithic configuration. All configurations were simulated with perfect branch prediction and memory disambiguation. The solid shaded bars on this graph are the IPC of the baseline clustered processor normalized by the IPC of the monolithic machine. The patterned bars represent the IPC improvement attained by using register caching.

The mod3 steering policy shows the most improvement in performance among the steering mechanisms. On average, register caching improves the

performance of mod3 steering by 6% for integer and floating point benchmarks. Load-slice and dependence steering policies show 1% IPC improvement.

We also examined the IPC improvement that can be obtained from register caching by simulating configurations with a Alpha 21264-like branch predictor. Figure 5.3 shows the IPC of the baseline clustered processor, with and without register caching, normalized by the IPC of the monolithic configuration. On average, the IPC of the integer benchmarks improved by 1% for mod3 steering and load-slice steering over their corresponding baselines. There was no change in the performance of the dependence steering policy. The floating-point benchmarks show an improvement of 1% for all three steering policies over their corresponding baselines.

The IPC improvements provided by register caching are very modest even though the register re-use statistics presented in Table 5.1 show that on average about 45% of all input operands are re-used. The reason for this incongruence is as follows. The operand re-use statistics presented in Table 5.1 do not take into account the distribution of instructions to clusters. While there is significant re-use of operands within the instruction stream, instructions are not always steered in a fashion that exploits re-use. In the example shown in Figure 5.1, register caching will benefit this code only if I4 is steered to the same cluster as I3. Suppose I4 is steered to cluster 3 instead of cluster 1 then both I3 and I4 will constitute first use of R24 in their respective clusters. In such a situation transfer instructions will be generated to transfer the value of R24 to clusters 1 and 3 and register caching will not help.

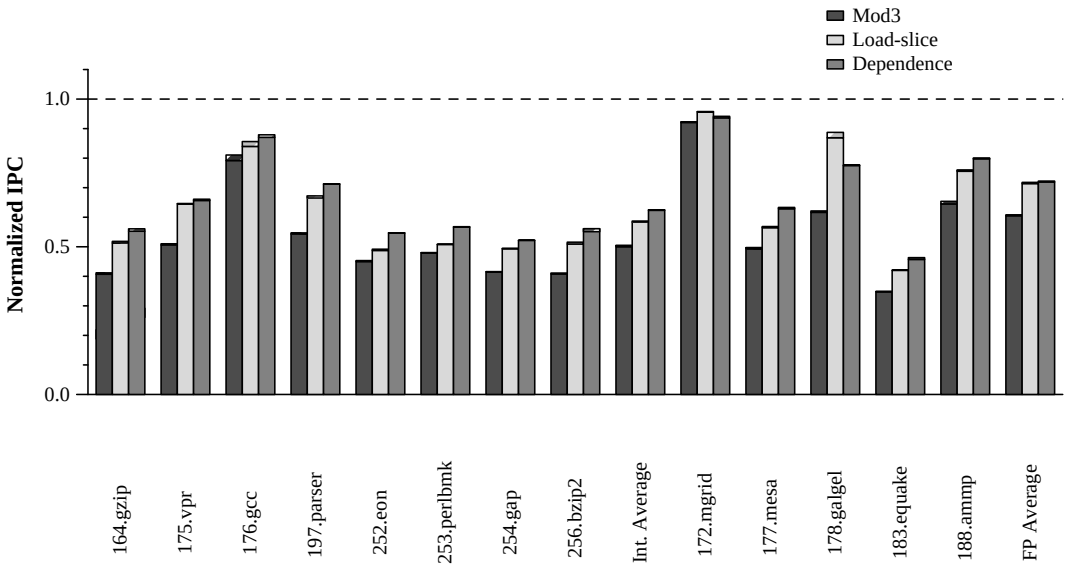

Figure 5.3: The IPC of a 16-wide 4-cluster processor, with and without register caching, normalized by the IPC of the ideal monolithic machine. Both the clustered processor and the monolithic machine configurations simulated Alpha 21264-like branch prediction and memory dependence prediction.

Register caching reduces the number of transfer instructions only marginally. Therefore, this method does not yield significant improvements in IPC. Even under the ideal conditions of perfect branch prediction and unlimited register cache capacity, IPC improves by only 6%. Our studies show that register caching is not a useful mechanism to reduce transfer instructions in clustered superscalar processors.

```
I1: Add R24, R32, R47     (R24 <-- R32 + R47)       Cluster 0
I2: And R56, R52, R47     (R56 <-- R47 && R52)      Cluster 0
I3: Add R127, R24, R125 (R127 <-- R24 + R125)       Cluster 1
I4: Sub R128, R24, R126 (R128 <-- R24 - R126)       Cluster 2
```

Figure 5.4: Example of a stream of instructions with inter-cluster dependence. Instructions I1 and I2 are assigned to cluster 0. Instruction I3 is assigned to cluster 1 and I4 to cluster 2.

5.2 Inter-cluster Operand Forwarding

In clustered superscalar processors, inter-cluster dependencies are detected in the rename stage of the pipeline. Once an inter-cluster dependence is detected a transfer instruction is inserted into the producing cluster to transfer the appropriate operand to the consuming cluster. For example, consider the fragment of code shown in Figure 5.4. The steering logic has assigned instructions I1 and I2 to cluster 0 and I3, which is dependent on I1, to cluster 1. In the rename stage of the pipeline this inter-cluster dependence is detected and a transfer instruction is inserted in cluster 0 to copy the value of R24 after I1 has executed.

Transfer instructions, as in the above example, are generated for every inter-cluster dependence that is detected. Though they facilitate inter-cluster communication, transfer instructions can be a significant overhead. We propose to replace transfer instructions with hardware signals. In this section we describe and evaluate two methods that can be used to remove transfer instructions—(a) consumer-requested forwarding and (b) hot-register based forwarding.

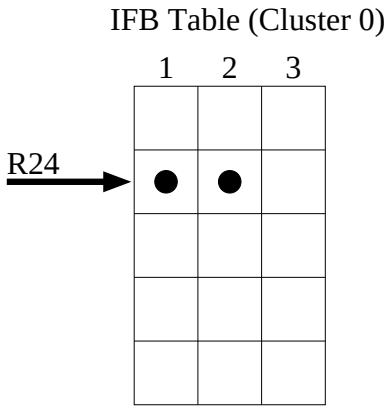

Figure 5.5: The inter-cluster forwarding bit table.

5.2.1 Consumer-requested Forwarding

In this mechanism, consumer instructions that require values from remote clusters explicitly request the value to be forwarded. For this purpose, each cluster is augmented with a register cache that is used to hold remote register values. We evaluated different register cache capacities and found that a 16-entry structure performed as well as a register cache that could hold all remote register values. The register caches use a simple FIFO replacement policy.

Each cluster is also augmented with a table that holds "inter-cluster forwarding bits" (IFB). These tables, shown in Figure 5.5, have an entry for each physical register within a cluster and for a processor with N clusters each entry is N-1 bits wide. The consumer-requested forwarding (CRF) mechanism works as follows. Instructions that require a remote register value set the appropriate bit in the remote cluster's IFB table. When any instruction

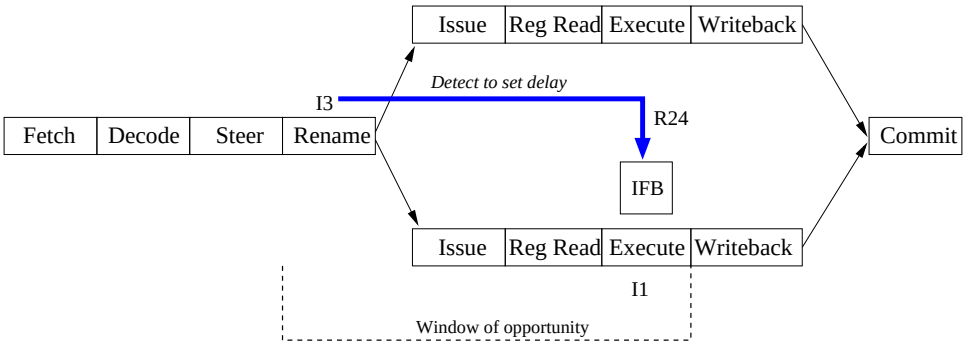

Figure 5.6: Clustered processor pipeline with inter-cluster forwarding bits.

producing a value is executed, the entry corresponding to the produced output register is read from the IFB table. This entry, a bit vector, will indicate whether instructions in other clusters require the produced value. The produced operand is forwarded to clusters whose bits have been set in the bit vector.

For example, consider the code segment shown in Figure 5.4, executing on a clustered processor illustrated in Figure 5.6. For simplicity we show only two clusters in the figure. In this example, when instruction I3 reaches the rename stage of the pipeline an inter-cluster dependence is detected. At this juncture, the baseline clustered processor will generate a transfer instruction to transfer the value of R24 from cluster 0 to cluster 1. However, in the CRF mechanism the transfer instruction is not generated. Instead the entry corresponding to R24 in cluster 0's IFB is set to indicate that the value R24 should be forwarded to cluster 1. Note that every entry in the IFB table is a bit vector. In this example, when I3 reaches the rename stage, the entry

in the vector corresponding to cluster 1 is set. Instruction I3 is then placed in the issue window of cluster 1 where it waits until both its source operands are ready. When I1 reaches the execute stage, the IFB table in cluster 0 is read to determine the clusters to which the result should be forwarded. By default, values are always forwarded to the local cluster. A wake-up signal is sent to instruction I3 in cluster 1 at the beginning of the cycle. Once I1 has finished execution the result is written to R24 and also forwarded to the register cache in cluster 1. Note that in the CRF mechanism both the wake-up tags and the result values are broadcast between clusters only if they are explicitly requested. If all other dependences are satisfied, I3 can be selected for issue. Once selected, instruction I3 reads R24 from the register cache and its other source operand from the register file in cluster 1 and proceeds to a functional unit to be executed. When the physical register R24 is eventually retired all cached copies are also invalidated. Figure 5.7 shows the pipeline timing diagram for the above example.

In the above discussion we assumed that producer (I1) reaches the execute stage after the consumer (I3) reaches the rename stage and has had sufficient time to set the bit in the IFB table. However, this situation does not always happen in practice. When instruction I3 reaches the rename stage of the pipeline there are three possible scenarios. The first case, instruction I1 has finished executing and had been committed. For such cases, where the producer has already exited the pipeline, the CRF method generates a transfer instruction to transfer the appropriate register value. In the second scenario,

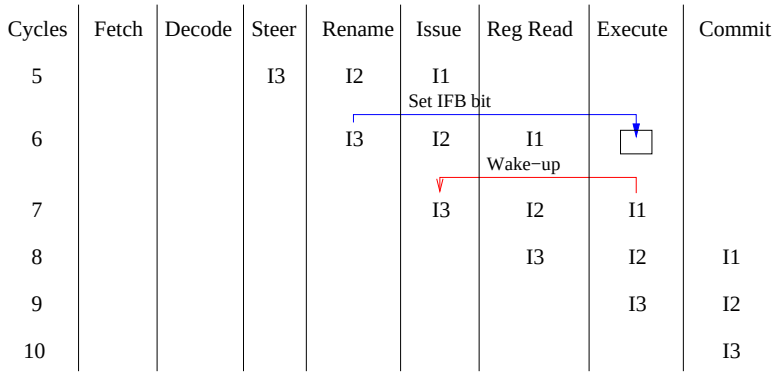

Figure 5.7: Pipeline timing for a clustered processor with consumer-requested forwarding.

instruction I1 is between the rename and execute stages of the pipeline and has not yet finished execution. In this case, instruction I3 sets the appropriate entry in the IFB table and proceeds to wait in cluster 1's issue window. When instruction I1 finishes execution, the IFB table is consulted and the value of R24 is forwarded to cluster 1. In the third scenario, instruction I1 has passed the execute stage of the pipeline by the time I3 reaches the rename stage. In this case, even though instruction I3 sets the IFB entry, the value of R24 is not forwarded since the IFB table is consulted only when I1 is in the execute stage of the pipeline. The situation described above leads to a pipeline deadlock since I3 will never get one of its input operands (R24). Furthermore, it may take multiple cycles between when an inter-cluster dependence is detected to when the corresponding IFB bit is set. We term this delay as the *detect-to-set* delay. The longer the detect-to-set delay the greater the probability of a deadlock.

As illustrated in Figure 5.6, consumer instructions have a "window of opportunity" within which to "catch" their producers. If the consumer instructions set the forwarding bits before the producer crosses the execute stage, the required remote value will be correctly forwarded. If the producer is past the execute stage before the forwarding bit is set then a deadlock occurs.

We propose two mechanisms that are used in conjunction with the CRF technique to avoid deadlocks. The first mechanism is called the *dual-wakeup* policy. In this mechanism every producer instruction checks the IFB table twice. Producer instructions read the IFB tables in the execute stage as described above. They access the IFB tables once more when they reach the commit stage, and wake up any dependent instructions and forward the required operand value. For the dual-wakeup policy to work all instructions must carry the value they produce along the pipeline until they exit the pipeline. The dual-wakeup mechanism avoids deadlocks but it delays consumer instructions that miss their window of opportunity.

The second method to avoid deadlock is called *pro-active operand fetch* (POF). In this technique consumer instructions that miss their window of opportunity obtain their remote operand values by reading it directly from the remote cluster's register file. The POF method works as follows. If the oldest instruction in a cluster's issue-window (e.g. I3) requires a remote operand the processor conservatively assumes that this instruction missed its window of opportunity. The hardware attempts to read the required operand value from the remote cluster. It will require a duration equal to twice the inter-cluster

communication delay to obtain the remote operand (round-trip communication delay). If the concerned operand had been produced in the remote cluster the pro-active operand read will succeed and I3 will be issued for execution. If the remote value has not yet been produced (i.e. the producer has not yet executed) the pro-active fetch will fail and the operand will be transferred after the producer executes. The POF technique eliminates deadlocks caused by consumers that miss their window of opportunity but it requires additional hardware to arbitrate between remote and local read accesses to the register file.

In addition to the deadlock situation discussed above there is one other corner case. Since the register caches are of finite capacity it is possible that a cached value is evicted before the dependent instruction that requires the value can be issued. When issued, the instruction will attempt to read its source operand from the register cache and will suffer a miss. We handle this corner case by first stalling the pipeline and then forwarding the required value to the instruction that caused the register cache miss. Normal execution is resumed once the value has been forwarded. An alternative mechanism to handle register caches is to flush the pipeline and restart execution from the instruction that caused the register cache miss. In our simulations we found that register cache misses are very infrequent and so they are not a bottleneck.

We evaluated the performance improvement that can be obtained over the baseline clustered processor using IFB bits in conjunction with the two techniques for avoiding deadlocks. In the rest of this section we present the

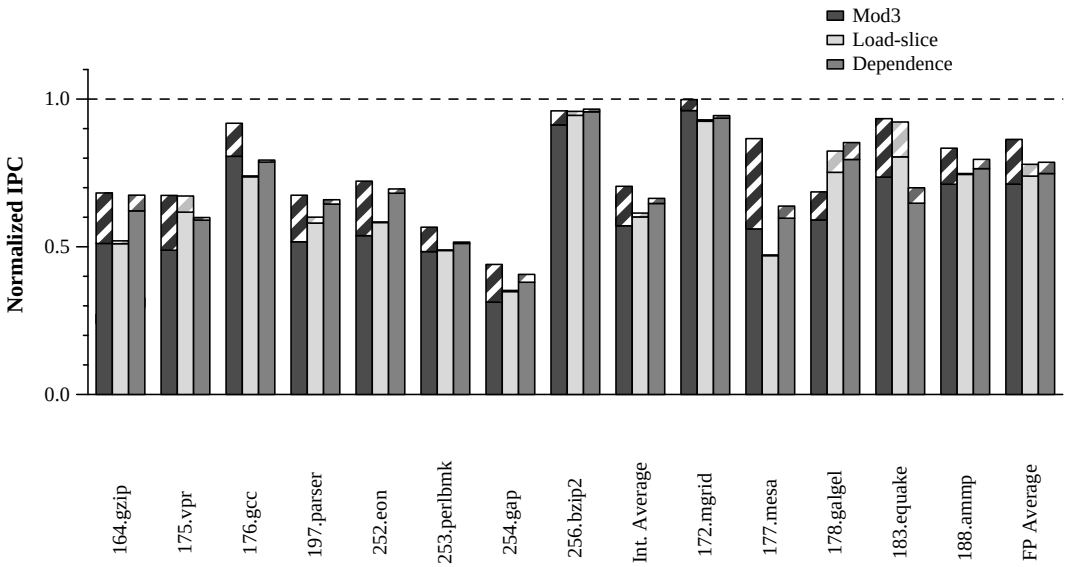

Figure 5.8: The IPC of a 16-wide clustered processor configuration, with and without the IFB mechanism, normalized by the IPC of the ideal monolithic machine. The configurations with the IFB mechanism used the dual-wakeup policy to avoid deadlocks. Both the clustered processor and the monolithic machine configurations simulated perfect branch prediction and perfect memory disambiguation.

results from our simulations.

5.2.1.1 Dual-wakeup

Using IFB tables to express inter-cluster dependence will significantly reduce transfer instructions and mitigate the effect of one of the bottlenecks in clustered processors. To quantify the effectiveness of IFB tables we evaluated the performance of SPEC 2000 benchmarks executing on a 16-wide, 4-cluster machine described in Section 3.3. We examined configurations with

and without perfect branch prediction. Figure 5.8 shows the IPC of the baseline clustered processor, with and without the IFB mechanism, normalized by the IPC of the monolithic configuration. All configurations shown in this figure simulated perfect branch prediction and perfect memory disambiguation. The solid shaded bars on this graph are the IPC of the baseline clustered processor normalized by the IPC of the monolithic machine. The patterned bars represent the IPC improvement that can be attained over the baseline by using the IFB mechanism. For these simulations we assumed a 1-cycle detect-to-set delay. Furthermore, these simulations used the dual-wakeup policy to avoid deadlocks.

Overall, using IFB tables improves performance of almost all benchmarks, and across all steering algorithms. On average, the IPC of the integer benchmarks improved by 25% for mod3 steering, 2% for load-slice steering and 3% for dependence steering over their corresponding baselines. The IPC of floating point benchmarks improved by 24% for mod3 steering, and by 5% for load-slice and dependence steering. In Section 4.2.1 we quantified the maximum performance that can be obtained by completely removing transfer instructions. Note that the IPC improvements obtained using IFB tables is close to that maximum value.

The above experiments assumed that the IFB table entries can be set with a 1-cycle detect-to-set delay. At small process technologies, and very aggressive clock frequencies multiple cycles may elapse between detecting an inter-cluster dependence and setting the appropriate IFB table entry. This

Steering	Int.			FP		
	2	3	4	2	3	4
mod3	7%	-14%	-18%	1%	-18%	-27%
ldslice	1%	-2%	-3%	4%	2%	0%
dependence	2%	0%	-1%	3%	-1%	-2%

Table 5.2: Average improvement the IPC of clustered processor with IFB tables compared to a baseline machine for increasing values of detect-to-set delay. All configurations were simulated with perfect branch prediction.

additional detect-to-set delay may cause many consumer instructions to miss their window of opportunity and therefore cause delayed wakeups. We quantified the sensitivity of IPC to this delay. Table 5.2 shows the average IPC improvement of an IFB-enabled machine over the baseline processor for increasing values of detect-to-set delay. As the detect-to-set delay increases a greater number of consumer instructions miss their window of opportunity resulting in an increased number of delayed wakeups. These delayed wakeups degrade performance. As the detect-to-set delay increases the benefit of IFB tables decreases. For detect-to-set delays of 3 cycle and greater the performance of the IFB enhanced processors is lower than the baseline processor.

We also performed experiments that simulated monolithic and clustered processor configurations with Alpha 21264-like branch and memory dependence predictors. Figure 5.9 shows the IPC of the baseline clustered processor, with and without the IFB mechanism, normalized by the IPC of the monolithic configuration. On average, using IFB tables improved the IPC of the integer benchmarks improved by 9% for mod3 steering, 3% for load-slice

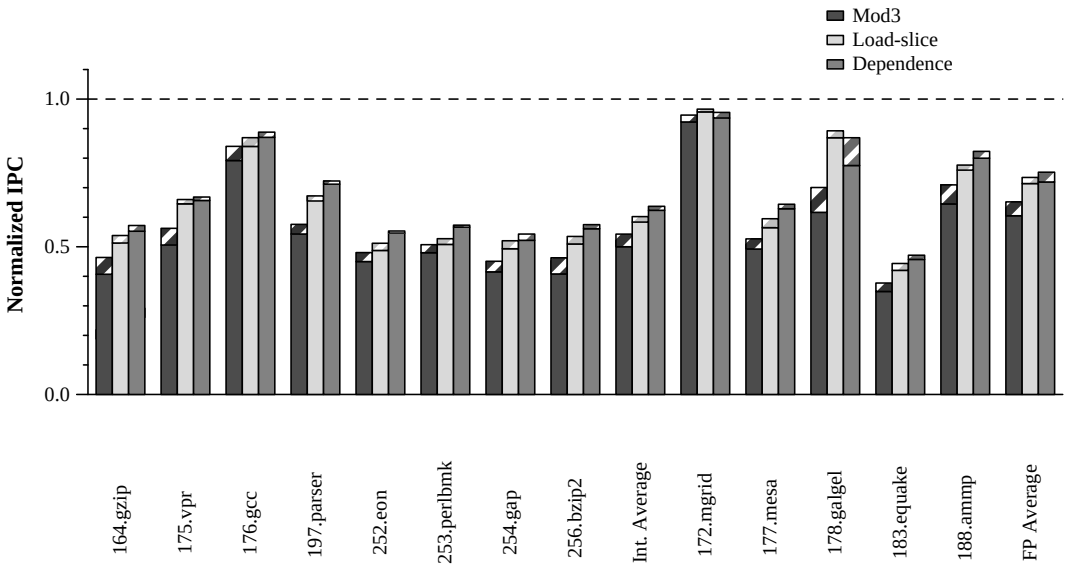

Figure 5.9: The IPC of a 16-wide clustered processor configuration, with and without the IFB mechanism, normalized by the IPC of the ideal monolithic machine. The configurations with the IFB mechanism used the dual-wakeup policy to avoid deadlocks. Both the clustered processor and the monolithic machine configurations used Alpha 21264-like branch prediction and memory dependence prediction.

steering and 2% for dependence steering over their corresponding baselines. The IPC of floating point benchmarks improved by 8% for mod3 steering, 3% for load-slice and 5% for dependence steering. In Section 4.2.1 we quantified the maximum performance that can be obtained by completely removing transfer instructions. Note that the IPC improvements obtained using IFB tables is close to that maximum value.

As explained in Section 4.2.1, the transfer instruction overhead is more significant when the pipeline is full of useful instructions. Therefore, the CRF

technique shows greater improvement for the experiments that simulate perfect branch prediction than for the ones that simulate configurations with a Alpha 21264-like predictor.

5.2.1.2 Pro-active Operand Fetch

We also examined the performance improvement that can be obtained by using IFB tables in conjunction with the pro-active operand fetch (POF) technique. We examined configurations with and without perfect branch prediction. Figure 5.10 shows the IPC of the baseline clustered processor, with and without the IFB mechanism, normalized by the IPC of the monolithic configuration. All configurations shown in this figure simulated simulated perfect branch prediction and perfect memory disambiguation. The solid shaded bars on this graph are the IPC of the baseline clustered processor normalized by the IPC of the monolithic machine. The patterned bars represent the IPC improvement that can be attained over the baseline by using the IFB mechanism. For these simulations we assumed a detect-to-set delay of 1 cycle. Furthermore, these simulations used the POF policy to avoid deadlocks.

On average, the IPC of the integer benchmarks improved by 27% for mod3 steering, 2% for load-slice steering and 3% for dependence steering over their corresponding baselines. The IPC of floating point benchmarks improved by 24% for mod3 steering, and by 5% for load-slice and dependence steering. Observe that for a detect-to-set latency of 1 cycle there is no difference in performance between the dual-wakeup and the pro-active fetch policies. We

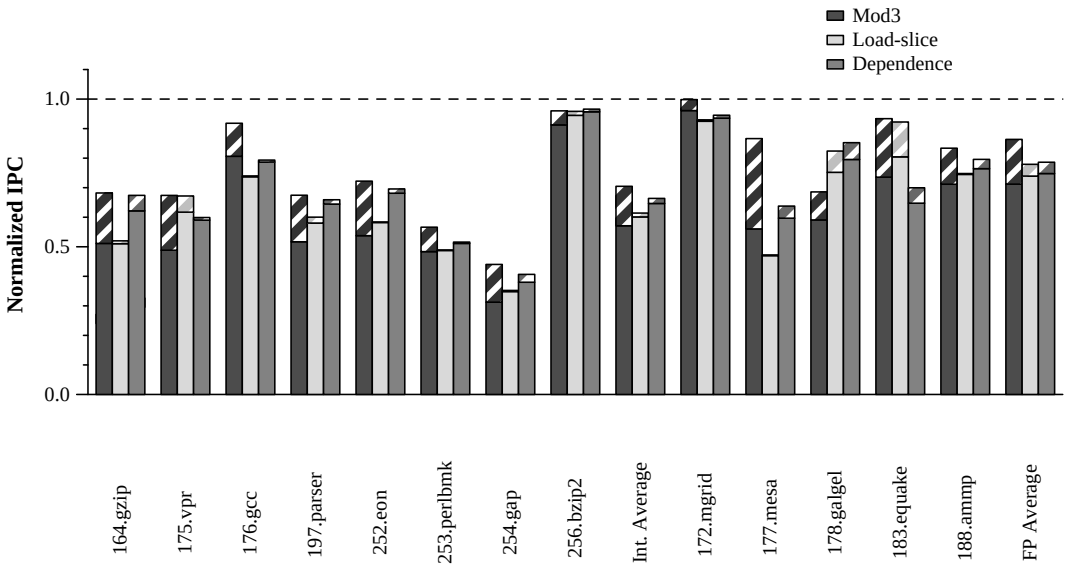

Figure 5.10: The IPC of a 16-wide clustered processor configuration, with and without the IFB mechanism, normalized by the IPC of the ideal monolithic machine. The configurations with the IFB mechanism used the pro-active operand fetch policy to avoid deadlocks. Both the clustered processor and the monolithic machine configurations simulated perfect branch prediction and perfect memory disambiguation.

also performed experiments that simulated monolithic and clustered processor configurations with Alpha 21264-like branch and memory dependence predictors. As in the perfect prediction case, there was no appreciable difference in performance between the dual-wakeup and the pro-active fetch policies.

The above experiments assumed that the IFB table entries can be set with a 1-cycle detect-to-set delay. Table 5.3 shows the average IPC improvement of an IFB-enabled machine over the baseline processor for increasing values of detect-to-set delay. Just as in the dual-wakeup policy, as the detect-

Steering	Int.			FP		
	2	3	4	2	3	4
mod3	8%	-13%	-17%	2%	-16%	-25%
ldslice	1%	-2%	-3%	4%	2%	1%
dependence	2%	0%	-1%	3%	-1%	-2%

Table 5.3: Average improvement the IPC of clustered processor with IFB tables compared to a baseline machine for increasing values of detect-to-set delay. All configurations were simulated with perfect branch prediction.

to-set delay increases the IPC improvement gained from using IFB tables decreases. For detect-to-set delays of 3 cycle and greater the performance of the IFB enhanced processors is lower than the baseline processor.

Both deadlock avoidance mechanisms show similar performance improvements over the baseline. However, the pro-active fetch policy requires clusters to be able to directly read values from remote register files. Additional hardware will be required to arbitrate between local and remote reads to the register files. The dual-wakeup policy is a better mechanism to avoid deadlock since it has lower design complexity.

5.2.2 Hot-register Based Forwarding

So far the techniques we have discussed to reduce transfer instructions rely on the consumer instructions explicitly signaling their producer instructions to forward operand values. In this section we describe a method that predicts the values to be forwarded based on past history. These predictions are made at the time the producer instruction reaches the rename stage of the

pipeline and so, unlike consumer-requested forwarding, this method does not have additional constraints such as the detect-to-set delay.

The hot-registers mechanism tracks the registers that are used by each cluster and uses this information to predict where instruction outputs should be forwarded. For this purpose a each cluster has a table, called the hot-register table, with one entry for every physical register in the processor. The hot-register prediction tables are located in the rename stage of the pipeline. Each entry in the table is a saturating counter that has three states—don't forward, forward, strong-forward. The hot-register tables in each cluster record the input operands of instructions executing on that cluster over the course of the program by incrementing the appropriate counter. This information is used to predict the clusters to which instruction outputs should be forwarded. The hot-register predictions may be incorrect. Sometimes a required value may not be forwarded to the correct cluster. Such a situation leads to a pipeline deadlock. We use the pro-active operand fetch policy, described in Section 5.2.1, to eliminate such deadlocks. When an required operand is not correctly forwarded the corresponding hot-register counter is incremented so that future instructions, producing that register value, forward the data correctly. Similarly, when a physical register is retired we check to see if the all forwarded copies of this register were used. If a forwarded value was not used by the cluster the corresponding hot-register entry is decremented. We call such unused values *frivolous forwards*. This mechanism ensures that future instructions producing that register value do not forward it to clusters that do

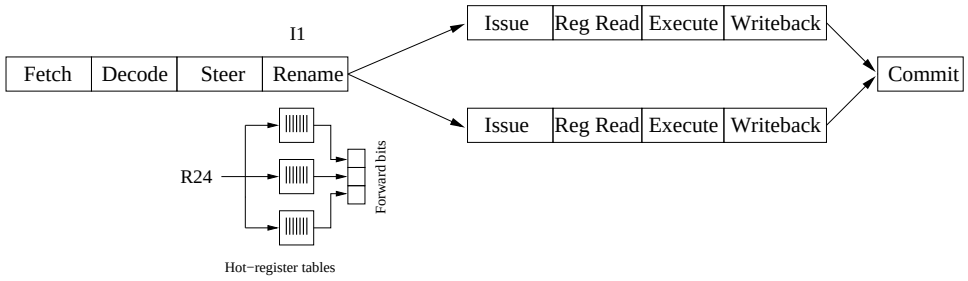

Figure 5.11: Clustered processor pipeline with hot-register based forwarding.

not require the value. A clustered processor pipeline with hot-register based forwarding is shown in Figure 5.11

The pipeline with hot-register tables functions as follows:

Step 1 At the rename stage, the hot-register table corresponding to the instruction's cluster is accessed. The counters in the table that correspond to the instruction's input operands (architectural registers) are incremented.

Step 2 The entry corresponding to the instruction's output architectural register is read from all hot-register tables. These values are combined to form a bit vector called "forward bits". Counters that are in the forward/strong-forward states contribute a 1 to this bit vector while counters in the don't forward state contribute a 0 to this bit vector. The forward-bits are a prediction made based on each cluster's operand utilization history. The forward-bits travel through the pipeline along with the instruction.

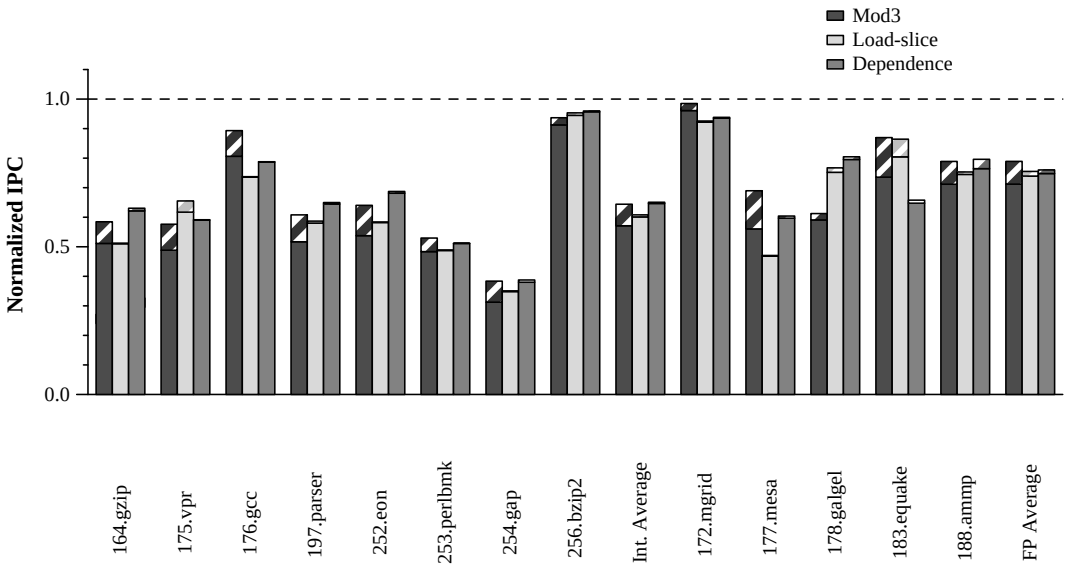

Figure 5.12: The IPC of a 16-wide clustered processor configuration, with and without the hot-register mechanism, normalized by the IPC of the ideal monolithic machine. Both the clustered processor and the monolithic machine configurations simulated perfect branch prediction and perfect memory disambiguation.

Step 3 After the instruction has finished execution its output value is forwarded to clusters identified in the forward-bits.

Step 4 When a physical register is retired we check to see if all of its forwarded values have been used. The corresponding hot-register counters in clusters that requested but did not consume this value are decremented.

To quantify the effectiveness of hot-register based prediction we evaluated the performance of SPEC 2000 benchmarks executing on a 16-wide, 4-cluster machine described in Section 3.3. We examined configurations with

and without perfect branch prediction. Figure 5.12 shows the IPC of the baseline clustered processor, with and without the hot-register mechanism, normalized by the IPC of the monolithic configuration. All configurations shown in this figure simulated simulated perfect branch prediction and perfect memory disambiguation. The solid shaded bars on this graph are the IPC of the baseline clustered processor normalized by the IPC of the monolithic machine. The patterned bars represent the IPC improvement that can be attained over the baseline by using the hot-register mechanism.

On average, the IPC of the integer benchmarks improved by 14% for mod3 steering, 1% for load-slice and dependence steering over their corresponding baselines. The IPC of floating point benchmarks improved by 12% for mod3 steering, and by 2% for load-slice and dependence steering. The hot-register forwarding mechanism shows improvement over the baseline but this improvement is lower than the CRF method with a 1-cycle detect-to-set delay. The hot-register mechanism predicts the cluster to which producer instructions should forward values and in some cases the predictions are incorrect. Such mis-predictions delay the execution of the corresponding dependent instructions and therefore the performance of the hot-register policy is lower than that of the CRF policy. Mis-predictions in the hot-register mechanism also result in frivolous forwards of data. We found that on average 35% of the forwarded values were frivolous forwards. Because of these frivolous forwards the hot-register mechanism will require a greater amount of power than the consumer-requested forwarding method.

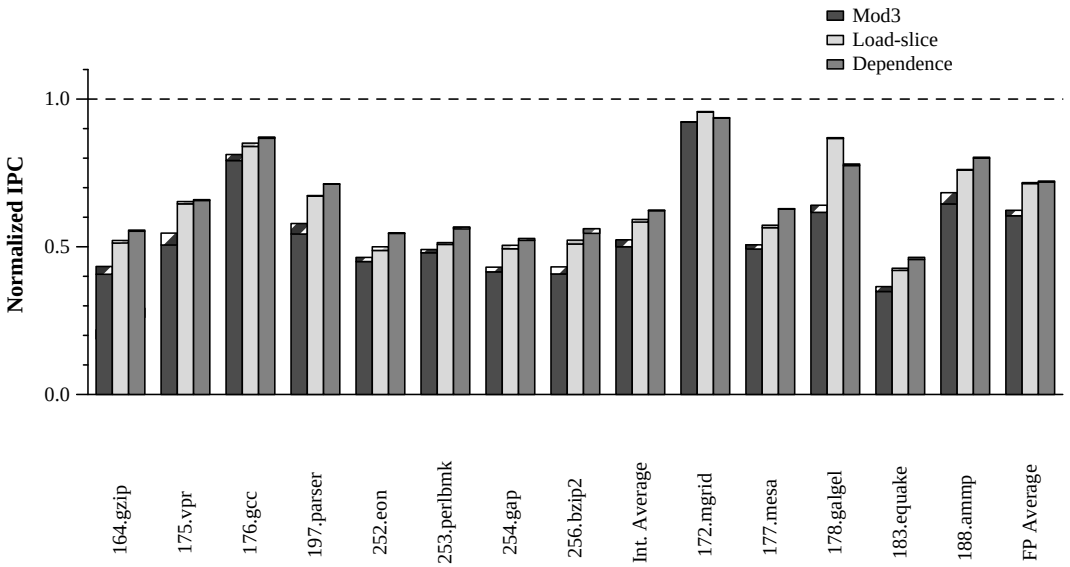

Figure 5.13: The IPC of a 16-wide clustered processor configuration, with and without the hot-register mechanism, normalized by the IPC of the ideal monolithic machine. Both the clustered processor and the monolithic machine configurations used Alpha 21264-like branch prediction and memory dependence prediction.

We also performed experiments that simulated monolithic and clustered processor configurations with Alpha 21264-like branch and memory dependence predictors. Figure 5.13 shows the IPC of the baseline clustered processor, with and without the IFB mechanism, normalized by the IPC of the monolithic configuration. For such a configuration, on average, the IPC of the integer benchmarks improved by 5% for mod3 steering, 2% for load-slice steering and 1% for dependence steering over their corresponding baselines. The IPC of floating point benchmarks improved by 3% for mod3 steering, and by 1% for load-slice and dependence steering.

The hot-register based forwarding mechanism improves the IPC of clustered machines by eliminating transfer instructions but this improvement is lower than that provided by consumer-requested forwarding with a 1-cycle detect-to-set delay. In addition, mis-predictions in the hot-register mechanism cause a large fraction of frivolous forwards and so result in greater power consumption. For these reasons the CRF mechanism is more effective than hot-register forwarding. However, technology scaling projections predict that the latency of wires will scale poorly. This poor scaling could result in increased detect-to-set delays in the consumer-requested forwarding method. In such a situation, a prediction based forwarding method, like the hot-register mechanism, will be preferable to consumer-requested forwarding. However, better prediction based mechanisms will have to be developed to reduce the number of frivolous forwards. Butts and Sohi proposed a method to predict the number of dynamic uses of a register value [13]. Such a predictor could be used to broadcast register values that have high degree of use to all clusters while generating transfer instructions for other values.

5.3 Related Work

In this chapter we examined the feasibility of caching remote register operands to reduce transfer instructions and to reduce the access latency to remote operands. Fast register access times are critical to high performance. Swensen and Patt evaluated a hierarchical register file consisting of a small set of fast access (1 cycle or less) registers and a larger set of slower access (multi

cycle) registers in a different context [68]. They observed that this hierarchical organization could perform as well as a large single-cycle register file.

Cruz *et al.* proposed a banked register file organization for superscalar processors [19]. Each register bank has a different access latency. They evaluated a multi-level register file organization. In this organization only the uppermost level can provide source operands to functional units. Results are always written into the lowest level and optionally into the upper levels. The upper level, in effect, subsets or caches the lower level register banks. Operand values that can be bypassed are written to only the lower register-bank while operands that cannot be bypassed are written to both register banks.

Both the above studies examined register caching as a means to reduce the access latency of register files in a un-clustered processor. Borch *et al.* describe a register caching mechanism for a clustered superscalar processor [9]. In their model there are eight functional unit clusters and a cluster register cache (CRC) associated with each cluster. In addition, there is a centralized register file. Instructions assigned to a cluster attempt to obtain their input operands from the CRC. On a CRC miss they access the required operand from the central register file. A structure, called the *insertion table*, is associated with each register cache. This structure is used to track the number of outstanding consumers of an operand that will execute on that cluster. Upon completing execution instructions forward their results to all CRCs and to the centralized register file. All values that are produced are written into the centralized register file but are written into a CRC only if the insertion

table entry associated with that value is non-zero. Their studies show that a CRC capacity of 16 entries with a FIFO replacement policy is close to the performance of a mechanism with perfect caching.

Brown and Patt proposed the demand-only broadcast technique to bypass values between clusters [11]. In this method an instruction's result is broadcast to remote clusters only if it is needed by dependent instructions in those clusters. For this purpose, an additional bit is added to the busy bit table (BBT) in every cluster. This bit, which we refer to as BBT-1, is used to indicate if the broadcast of a result should be blocked. Initially, all BBT-1 bits are set to 0, indicating that the corresponding register value should be blocked (i.e. not broadcast). When an instruction (I2) in cluster 2 that is dependent on another instruction (I1) in cluster 1 reaches the scheduling window it sets the BBT-1 entries corresponding to its source register operands. After I1 executes the result is forwarded to clusters cluster 2 since the corresponding BBT-1 bit in that cluster is set. The value produced by I1 will not be forwarded to other clusters that do not have the corresponding BBT-1 entry set.

The demand-only broadcast method is similar to the consumer requested forwarding technique discussed in Section 5.2.1. Both methods require the approximately same amount of hardware state. The the IFB tables in the CRF method have one entry per physical register and each entry is 3 bits wide. The demand-only broadcast policy requires an additional bit in the BBT tables in each cluster. The number of entries in a BBT table is equal to the number of physical registers in the machine. The main difference between

the two techniques is that in the demand-only broadcast method the wake-up tags are always broadcast to all clusters. In the CRF method, wake-up tags are broadcast only on-demand. Therefore, the CRF policy requires a fewer number of ports in the instruction issue window to broadcast wake-up tags. However, selectively broadcasting wake-up tags could potentially deadlock the processor. The CRF mechanism requires the dual-wakeup or the pro-active fetch mechanism discussed in Section 5.2.1 to avoid deadlocks.

5.4 Summary

In this chapter we discussed several techniques to remove transfer instructions. The first method that we examined attempts to reduce the number of transfer instructions by caching remote register values. This mechanism eliminates multiple transfer instructions from being generated for re-used operands. Our evaluation showed that while there is moderate re-use of operands within the instruction stream, instructions are not steered in a fashion that exploits this re-use. Therefore, the register caching mechanism provides little improvement in performance.

We also examined two techniques that replace transfer instructions with hardware signals—(a) consumer-requested forwarding (CRF) and (b) hot-register based forwarding. In the CRF method, consumer instructions that require values from remote clusters explicitly request the value to be forwarded by setting a bit in a structure called the inter-cluster forward bit table. Producer instructions, upon completion of execution, consult this table

to determine if the value should be forwarded to other clusters. The IPC improvement from the CRF method is very close to the maximum improvement that can be gained by completely eliminating all transfer instructions.

Inter-cluster dependences between instructions are detected when the consumer instruction reaches the rename stage of the pipeline. When such a dependence is detected the corresponding forward bit is set. We term the time between detecting the dependence to setting the forward-bit as the detect-to-set delay. We found that the benefits of the CRF policy are reduced as the detect-to-set delay increases. For detect-to-set latencies of 3 cycles and greater the CRF mechanism provides no performance improvement.

The hot-registers mechanism tracks the registers that are used by each cluster and uses this information to predict where instruction outputs should be forwarded. These predictions are made at the time the producer instruction reaches the rename stage of the pipeline. Though the hot-register forwarding mechanism shows improvement over the baseline this improvement is lower than the CRF method with a 1-cycle detect-to-set delay. Incorrect forwarding-predictions in the hot-register mechanism delay the execution of the corresponding dependent instructions and therefore the performance of this policy is lower than that of the CRF policy. Hot-register forwarding is a prediction based mechanism and does not have additional constraints like the detect-to-set delay in the CRF mechanism. Technology scaling projections suggest that the latency of wires will scale poorly. This poor scaling could result in increased detect-to-set delays at future technologies. If the

detect-to-set delay is greater than 2 cycles then the CRF mechanism is no longer useful. In such a situation, prediction based forwarding mechanisms, like the hot-register mechanism, will perform better than consumer-requested forwarding. However, the hot-register mechanism generates a large number of frivolous forwards which results in greater power consumption. More accurate prediction based mechanisms will have to be developed to reduce the number of frivolous forwards.

The inter-cluster forwarding techniques discussed in this chapter yield IPC improvements from 1% to of 9% over the baseline processor. Such small gains in performance are not worth the additional design complexity these mechanisms impose. Transfer instructions will become a significant bottleneck only if the pipeline is frequently full with useful instructions. Therefore, these techniques will be useful only if researchers improve control and memory prediction significantly.

Chapter 6

Instruction Steering

In Chapter 5 we discussed techniques to remove transfer instructions. In this chapter we propose techniques to address the other two bottlenecks in clustered processors—cluster resource limitations and inter-cluster communication. We propose new steering policies to mitigate the effect of these bottlenecks.

Instruction steering policies for clustered processors must attempt to reduce inter-cluster communication penalties and provide a balanced distribution of instructions to clusters. However, there is a trade-off between these two objectives and optimizing for just one of the two problems exacerbates the other. The steering mechanisms that we have discussed so far—mod3, load-slice, and dependence—are at different extremes in the spectrum of steering mechanisms. The mod3 steering mechanism tries to maximize the utilization of processor resources but pays no heed to inter-cluster communication. As shown in Section 4.2.3, this policy has fewer structure capacity stalls and issue-limited instructions compared to load-slice and dependence steering. Dependence steering, on the other hand, attempts to reduce inter-cluster communication penalties by steering all dependent instructions to the same cluster

but completely ignores resource utilization. Similarly, the load-slice steering policy steers all instructions that are part of the address computation for a load instruction to the same cluster. This policy also optimizes for inter-cluster communication alone while ignoring instruction load balance. However, both dependence and load-slice steering reduce inter-cluster communication significantly. For these steering policies, on average, 8% of source operands are read from remote clusters while for mod3 steering 66% of source operands are remote operands. In this chapter we propose and evaluate steering policies that attempt to find a balance between inter-cluster communication and processor utilization.

As discussed in Chapter 4, the effect of the clustered processor bottlenecks varies depending on how full the processor pipeline is kept. In this chapter we evaluate our new steering policies by simulating configurations with both an Alpha 21264-like branch predictor and with perfect branch prediction.

6.1 Memory Instruction Steering

Some of the steering mechanisms that we have explored so far, such as dependence and load-slice steering, attempt to reduce inter-cluster operand communication. However, none of these policies attempt to address the latency of memory instructions. The baseline clustered processor, described in Section 3.3, partitions the level-1 data-cache among the clusters. Each cluster has direct access to a 16KB cache bank. Cache lines are statically interleaved among the cache banks based on the low-order address bits. In this organiza-

tion consecutive cache-lines in memory will map to different banks. Note that a given address will always map the same cache bank.

The level-1 data cache can be organized in several other configurations. For example, cache lines in memory could be dynamically mapped to any of the level-1 banks. Designers may also choose to allow replication of data between the cache banks. While dynamic mapping and data replication may increase performance over a statically mapped organization they also increase design complexity. In this study we evaluate our ideas in the context of a simple statically mapped organization. Agarwal examined dynamic mapping of data to cache banks and data replication across the banks [1].

Since the baseline processor partitions the level-1 data cache, some memory instructions that are steered to one cluster (cluster 0) may require access to cache banks in other clusters (cluster 1). We refer to such a memory access as a *remote cache access*. Remote memory accesses take longer than local accesses. The additional latency is equal to twice the communication delay between the clusters (round-trip delay). Since the inter-cluster communication delay could be 1 or 2 cycles (see Section 3.3), the communication latency for a remote memory access could be 2 or 4 cycles.

Table 6.1 shows the number of remote cache accesses as a fraction of all cache accesses for the three baseline steering mechanisms. A significant fraction of memory instructions require remote cache accesses for all three steering policies. On average, about 75% of all memory instructions in the baseline clustered architecture access remote cache banks. In a monolithic

Benchmark	Mod3	Load-slice	Dependence
164.gzip	74%	84%	67%
175.vpr	74%	74%	73%
176.gcc	75%	67%	68%
197.parser	75%	75%	72%
252.eon	75%	75%	76%
253.perlbmk	75%	80%	79%
254.gap	74%	73%	74%
256.bzip2	74%	83%	73%
172.mgrid	75%	77%	74%
177.mesa	74%	82%	79%
178.galgel	74%	74%	75%
183.equake	75%	70%	75%
188.ammp	75%	73%	75%
Average	75%	76%	74%

Table 6.1: Remote cache accesses as a fraction of the total number of memory instructions executed by the baseline clustered processor.

machine, memory instructions access one unified cache and on a cache-hit the corresponding value is forwarded to dependent instructions in 3-cycles (cache hit latency). But in a clustered processor, a large fraction of memory instructions pay an additional communication delay before the value is forwarded to dependent instructions. Such remote cache accesses in clustered processors are one reason for the degradation in their IPC compared to a monolithic machine.

In this chapter we propose a memory steering policy that attempts to reduce the number of remote memory accesses by steering memory instructions to clusters that can obtain the required value from their local cache bank. The memory steering policy is overlayed on top of the baseline steering policies and works as follows. Non-memory instructions are steered based on the baseline

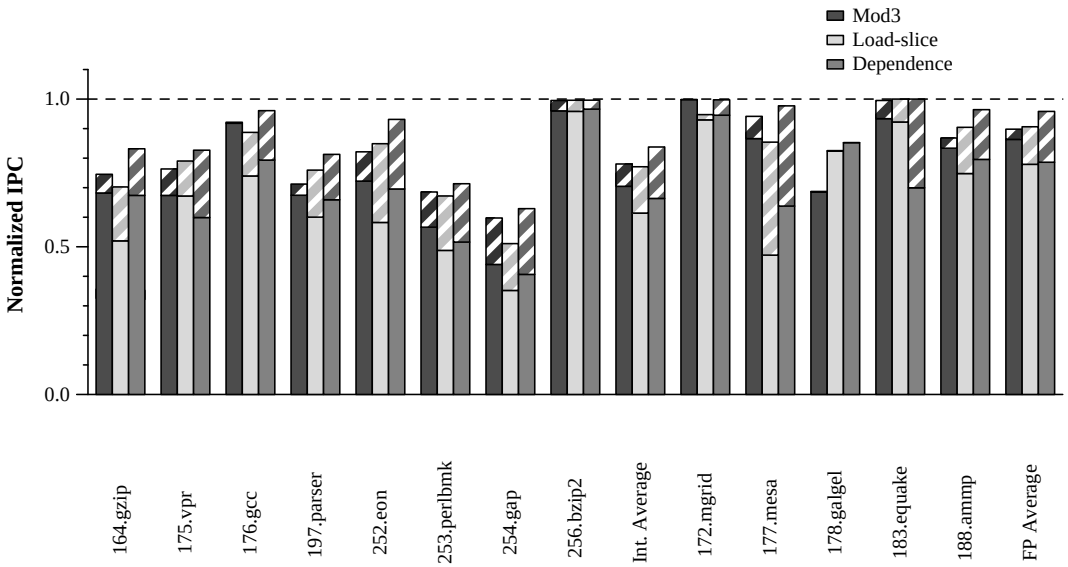

Figure 6.1: The IPC of a 16-wide clustered processor configuration, with and without *ideal* memory steering, normalized by the IPC of the ideal monolithic machine. Both the clustered processor and the monolithic machine configurations simulated perfect branch prediction and perfect memory disambiguation. Also, the configurations in these experiments used the CRF method to remove transfer instructions.

steering policy (i.e. mod3, load-slice or dependence) but when a memory instruction reaches the steer stage of the pipeline we predict the cache bank that this instruction will access. The instruction is then steered to the cluster with local access to the predicted cache bank.

6.1.1 Ideal Memory Instruction Steering

To quantify the maximum improvement that can be obtained using the memory steering policy we simulated a configuration where every memory instruction is steered to the "correct cluster" (i.e. a cluster that has local access

to the required value). We call this configuration *ideal memory steering*. These simulations used the CRF mechanism with a 1 cycle detect-to-set latency, described in Section 5.2.1, to orchestrate inter-cluster communication.

Figure 6.1 shows the IPC of the baseline clustered processor, with and without ideal memory steering, normalized by the IPC of the monolithic configuration. All configurations shown in this figure simulated perfect branch prediction and perfect memory disambiguation. The shaded bars on this graph are the IPC of the baseline clustered processor normalized by the IPC of the monolithic machine. The patterned bars represent the IPC improvement that can be attained over the baseline if memory instructions are steered to clusters in a fashion that ensures the instruction will not require a remote cache access. On average, ideal memory steering improved the IPC of integer benchmarks by 13% for mod3 steering, and by 29% for load-slice and dependence steering. The IPC of floating point benchmarks improved by 4% for mod3 steering, 23% for load-slice and 24% for dependence steering.

Overlaying memory steering on top of dependence or load-slice steering changes the resulting instruction distribution significantly. For example, consider the data dependence graph shown in Figure 6.2. The nodes in this graph represent instructions and the edges represent data dependence. The shaded nodes represent memory instructions. In this example instructions I2-I9 are all dependent on instruction I1 (either directly or through other nodes). Therefore, conventional dependence steering will steer all these instructions to the same cluster as I1. On the other hand, when dependence steering is used

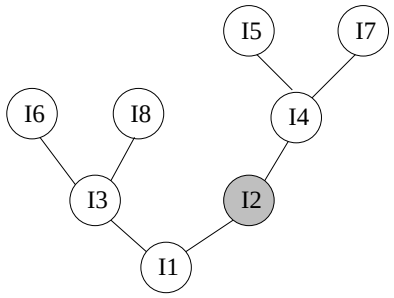

Figure 6.2: An example data dependence graph. The nodes represent instructions and the edges represent dependence. The shaded node represents a memory instruction.

in conjunction with memory steering the load instruction I2 is steered based on memory steering rather than dependence steering. This instruction may be placed in a different cluster from I1. In that case, the instructions that are dependent on I2 will also be steered to the same cluster as I2. In effect, I2 and its data dependence sub-tree could be steered to one cluster while I1 and its left sub-tree (comprising of I3, I6 and I8) could be steered to another cluster. For this reason, memory steering produces a more balanced instruction distribution compared to dependence and load-slice steering.

We evaluated the workload imbalance in the following manner. Every cycle we rank order the clusters based on the number of instructions in each cluster (cluster with most instructions has highest rank). We track the number of instructions issued from every rank. Table 4.6 shows the fraction of instructions that are executed by each rank for a 16-wide, 4-cluster machine with memory steering overlayed on dependence steering. For most benchmarks memory steering produces a better balance of instructions compared

Benchmark	Rank 0	Rank 1	Rank 2	Rank 3
164.gzip	0.19	0.22	0.26	0.33
175.vpr	0.22	0.24	0.24	0.30
176.gcc	0.25	0.26	0.24	0.25
197.parser	0.24	0.23	0.24	0.29
252.eon	0.20	0.27	0.27	0.26
253.perlbmk	0.22	0.25	0.28	0.26
254.gap	0.25	0.23	0.24	0.27
256.bzip2	0.21	0.25	0.26	0.29
172.mgrid	0.22	0.23	0.26	0.29
177.mesa	0.23	0.23	0.26	0.28
178.galgel	0.23	0.23	0.27	0.27
183.equake	0.24	0.21	0.28	0.27
188.ammp	0.24	0.28	0.23	0.25

Table 6.2: Workload imbalance in a 16-wide, 4-cluster machine using memory steering overlayed on dependence steering.

to dependence steering (Table 4.6, Section 4.2.3). A balanced instruction distribution results in fewer resource conflicts, such as structure capacity stalls and issue-limited instructions, compared to dependence and load-slice steering. Therefore, memory steering performs better than both dependence and load-slice steering. Furthermore, memory steering (overlayed on dependence or load-slice steering) requires fewer remote operands compared to mod3 steering and so it also out performs mod3 steering.

Overlaying, memory steering on top of dependence and load-slice steering has two prominent effects. The first effect is to produce a more balanced instruction distribution as discussed above. The second effect is to lower the number of remote cache accesses. These two effects together account for the

improvements shown in Figure 6.1.

6.1.2 Memory Instruction Steering with Last-cluster Prediction

So far we have evaluated the maximum IPC improvement that is possible from using memory steering. For these experiments we simulated configurations wherein memory instructions are always steered to a cluster that can access the required data from a local bank. In practice, it is not possible to steer all memory instructions to the "correct" cluster since the address that they will access may not be known at steer time. When a memory instruction reaches the steer stage we predict the cache bank that it will access. We use a simple prediction mechanism that steers instructions to the last cache bank that they accessed. For this purpose, a table, called the *last-cluster table*, is used to track the cache bank accessed by every memory instruction. This table is indexed by the memory instruction's program counter. When a memory instruction reaches the steer stage of the pipeline the last-cluster table is consulted to determine the cluster that the instruction should be steered to. When memory instructions complete execution they update the last-cluster table with the cluster that they accessed. As described above, the memory steering policy is used in conjunction with the baseline steering mechanisms. Memory instructions are steered using memory steering while other instructions are steered using the corresponding baseline policy.

We evaluated the effectiveness of the memory steering policy with the last-cluster prediction mechanism by simulating a 16-wide, 4-cluster proces-

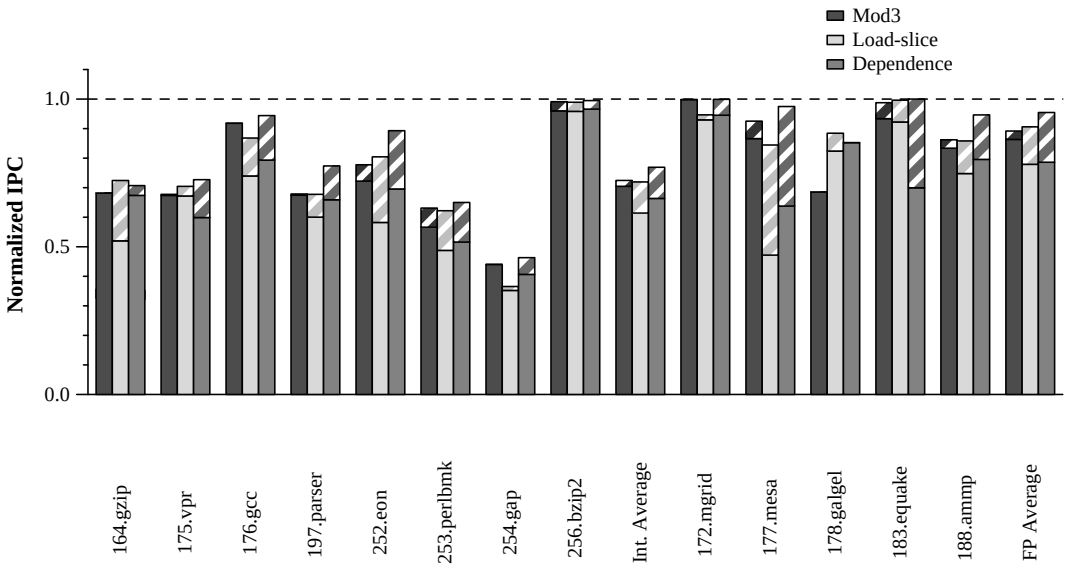

Figure 6.3: The IPC of a 16-wide clustered processor configuration, with and without last-cluster memory steering, normalized by the IPC of the ideal monolithic machine. Both the clustered processor and the monolithic machine configurations simulated perfect branch prediction and perfect memory disambiguation. Also, the configurations in these experiments used the CRF method to remove transfer instructions.

sor. We simulated configurations with perfect and Alpha 21264-like branch prediction. In addition, these simulations used the CRF mechanism with a 1 cycle detect-to-set latency, described in Section 5.2.1, to orchestrate inter-cluster communication. We varied the capacity of the last-cluster table and found that a 4096-entry table has the same prediction accuracy as a table with a unique entry for every memory instruction in the program. For a few benchmarks the prediction accuracy dropped by 10% for a 2048-entry table. Therefore, in these simulations we used a last-cluster prediction table with 4096 entries. Figure 6.3 shows the IPC of the baseline clustered processor,

with and without memory steering using the last-cluster table, normalized by the IPC of the monolithic configuration. All configurations shown in this figure simulated perfect branch prediction and perfect memory disambiguation. The shaded bars on this graph are the IPC of the baseline clustered processor normalized by the IPC of the monolithic machine. The patterned bars represent the IPC improvement that can be attained over the baseline using memory steering. On average, the last-cluster memory steering mechanism improves the performance of integer benchmarks by 3% for mod3 steering, 18% for load-slice steering, and 17% for dependence steering. The IPC of floating point benchmarks improved by 3% for mod3 steering, 22% for load-slice, and 24% for dependence steering.

As discussed earlier, there are two factors that improve IPC when using memory steering—improved workload balance and fewer remote cache accesses. Table 6.3 shows the number of remote cache accesses as a fraction of the total number of memory instructions executed by the clustered processor utilizing memory steering in conjunction with the baseline steering policies. The simple last-cluster prediction mechanism reduces the number of remote cache accesses for all benchmarks but its effectiveness varies across benchmarks. For a few benchmarks, namely 256.bzip2, 177.mesa, and 183.equake, there is a large reduction in the number of remote cache accesses. For these benchmarks, the number of remote cache accesses is reduced to less than 20% of all memory instructions. Other benchmarks, such as 164.gzip, 252.eon, 253.perlbmk, show an appreciable reduction in the number of remote cache accesses. For

Benchmark	Mod3		Load-slice		Dependence	
	Mem	Base	Mem	Base	Mem	Base
164.gzip	38%	74%	36%	84%	35%	67%
175.vpr	54%	74%	54%	74%	53%	73%
176.gcc	69%	75%	64%	67%	64%	68%
197.parser	56%	75%	55%	75%	55%	72%
252.eon	44%	75%	42%	75%	43%	76%
253.perlbmk	36%	75%	32%	80%	33%	79%
254.gap	64%	74%	63%	73%	63%	74%
256.bzip2	14%	74%	11%	83%	10%	73%
172.mgrid	67%	75%	67%	77%	66%	74%
177.mesa	15%	74%	11%	82%	12%	79%
178.galgel	73%	74%	73%	74%	73%	75%
183.equake	18%	75%	18%	70%	16%	75%
188.ammp	40%	75%	40%	73%	37%	75%
Average	45%	75%	44%	76%	43%	74%

Table 6.3: Remote cache accesses as a fraction of the total number of memory instructions executed by the baseline clustered processor using memory steering (Mem) and with the baseline steering policies (Base).

these benchmarks, the last-cluster prediction mechanism reduces the fraction of remote cache access from 75% (see Table 6.1) to less than 45%. For another set of benchmarks, namely 156.vpr, 176.gcc, 197.parser, 254.gap, 172.mgrid, and 178.galgel, the last-cluster prediction mechanism reduces the number of remote cache accesses only by a small amount. The IPC improvements of individual benchmarks correlates with the effectiveness of the last-cluster prediction mechanism for that benchmark. The load-slice and dependence steering policies result in a skewed workload distribution among clusters. Using memory steering in conjunction with these policies results in a better workload distribution in addition to reducing the number of remote cache accesses.

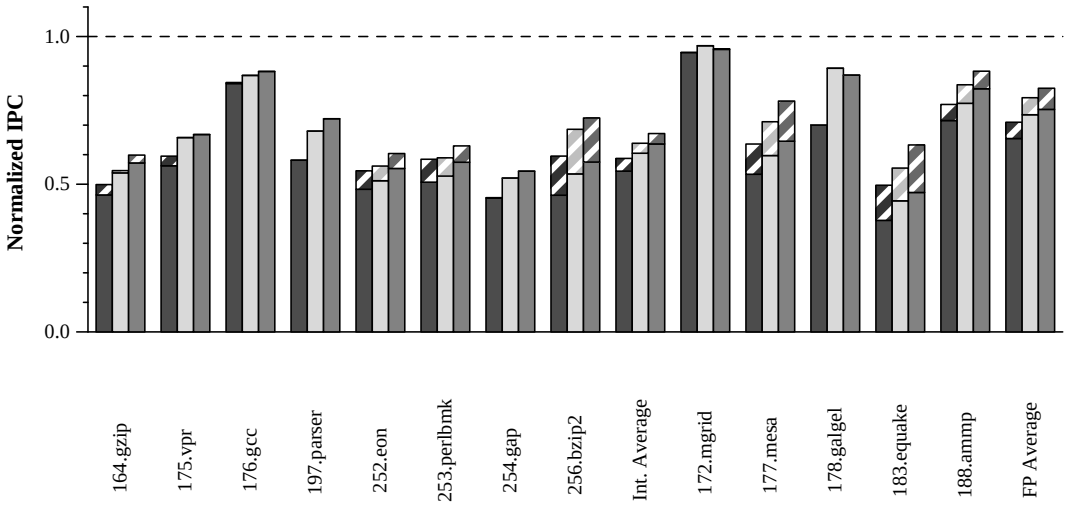

Figure 6.4: The IPC of a 16-wide clustered processor configuration, with and without last-cluster memory steering, normalized by the IPC of the ideal monolithic machine. Both the clustered processor and the monolithic machine configurations used Alpha 21264-like branch prediction and memory dependence prediction. Also, the configurations in these experiments used the CRF method to remove transfer instructions.

Therefore, the load-slice and dependence steering baselines, when used in conjunction with memory steering, show a greater IPC improvement than the mod3 policy.

We also examined the IPC improvement that can be obtained from last-cluster based memory steering by simulating configurations with Alpha 21264-like branch and memory dependence predictors. Figure 6.4 shows the IPC of an 16-wide, 4-cluster machine with the different steering policies normalized by the IPC of a monolithic machine. On average, the last-cluster memory steering mechanism improves the performance of integer benchmarks by 9%

for mod3 steering, 5% for load-slice steering, and 5% for dependence steering. The IPC of floating point benchmarks improved by 12% for mod3 steering, 10% for load-slice, and 12% for dependence steering. Memory steering with last-cluster prediction shows improvement over the mod3 steering for all benchmarks. This policy also shows an improvement over load-slice and dependence steering for most benchmarks. However, for a few benchmarks–197.parser, 175.vpr, 254.gap, and 178.galgel—the memory steering policy shows lower IPC compared to dependence and load-slice steering. For these benchmarks the accuracy of the last-cluster based cache-bank predictor is poor and therefore it does not reduce the number of remote cache accesses. In addition, as discussed earlier, using memory steering in conjunction with dependence and load-slice steering results in a instruction distribution that requires more inter-cluster (register) operand communication. Memory steering increased the amount of inter-cluster operand communication by a factor of 2.5.

For the benchmarks listed above, the last-cluster prediction mechanism does not reduce the number of remote cache accesses significantly and at the same time it increases the number of remote operands. Therefore, for these benchmarks last-cluster based memory steering reduces performance compared to the baseline.

In addition to the last-cluster prediction method we also examined the feasibility of using a stride-based cache bank predictor to steer memory instructions. Like the last-cluster predictor the stride predictor uses a table to record the cache bank that a memory instruction accessed. In addition, the

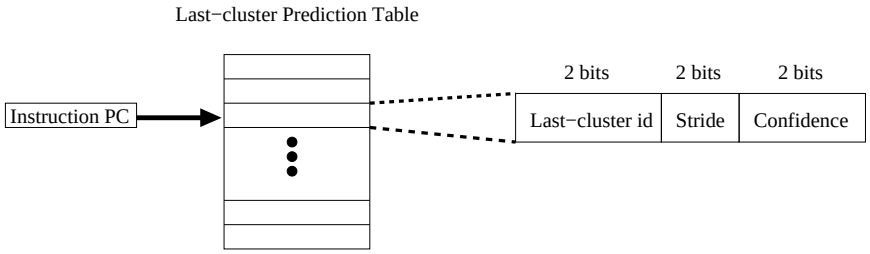

Figure 6.5: Memory instruction steering with a cluster-stride predictor.

table also records the "stride" (i.e. difference in cluster identity numbers) between the last cache bank the instruction accessed and the current cache bank. Figure 6.5 shows the cache bank-stride prediction table. When memory instructions reach the steer stage they read the stride predictor by indexing into the table using the instruction PC. If the confidence in the stride is low, the instruction is steered to the same cluster that was accessed the last time. If the prediction confidence is high, the instruction is steered to the cluster that is numerically a "stride" distance away. Like in the last-cluster method, memory instructions update the stride prediction table after execution. The stride predictor reduces the number of remote cache accesses to 41% for 197.parser and 45% for 254.gap and improves their IPC by 2% and 5% respectively.

The last-cluster based prediction and the stride prediction mechanisms are simple methods to predict the cache bank that a load or store instruction will access. While their prediction accuracies are high for several benchmarks, they do not work as well for 197.parser, 175.vpr, 254.gap and 178.galgel. These benchmarks have cache-bank access patterns that cannot be captured using a

simple last-cluster mechanism or a stride based mechanism. Other predictors, based on bank access pattern history, could be used to improve the prediction accuracy. Yoaz *et al.* proposed using prediction mechanisms based on bank access history [76]. Also, many techniques have been proposed to predict the address accessed by memory instructions [4, 8, 18]. Such techniques could be used to steer memory instructions to the correct cluster. Increasing the cache-bank prediction accuracy using more sophisticated predictors will further reduce remote cache accesses and thereby benefit these benchmarks.

6.2 Critical Operand Steering

The mod3 steering and dependence steering mechanisms both have advantages and a few disadvantages. The mod3 policy spreads instructions evenly across clusters and thereby making maximum use of the processor's issue window capacity and issue width. However, this policy also results in a large number of instructions requiring remote register operands. The dependence steering policy steers dependent instructions to the same cluster. If the two source operands of an instruction are being produced by different clusters the instruction is steered to the producer cluster with a fewer number of instructions (i.e. the cluster with the lighter load). The dependence steering policy steers too many instructions to one cluster and therefore does not make complete use of the processor resources like issue-bandwidth and issue window capacity.

In this section we propose and evaluate the critical-operand steering

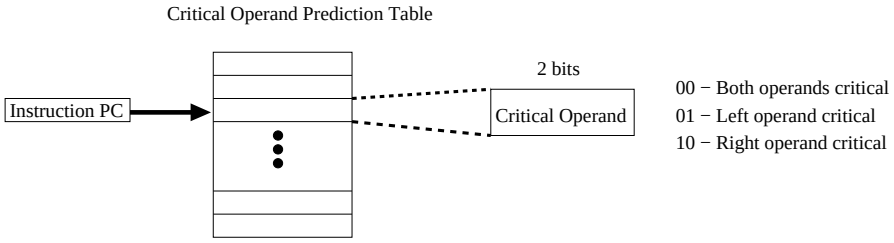

Figure 6.6: Critical-operand prediction table.

policy that attempts to find a balance between processor resource utilization and inter-cluster communication delay. This steering policy works as follows. When instructions reach the steer stage of the pipeline we predict which of its two source operands is more critical and steer the instruction to the cluster that produces the more critical operand. For this purpose we use a hardware structure called the *critical operand table*. The critical-operand table, shown in Figure 6.6, is indexed by the instruction PC. Each entry in the table is 2-bits wide. For each instruction we track which of its two source operands was produced last (i.e. the source operand that delayed the issue of the instruction). We deem this operand to be the critical operand for this instruction. For every instruction, the critical operand table is updated with information on which of the instruction's two source operands is critical. When instructions reach the steer stage of the pipeline they access the critical operand table to determine which of their source operands is more critical. The instruction is then steered to the same cluster as the producer of the more critical operand. In the event that both its operands are equally critical the instruction is then steered to the producer cluster with the lightest load. We experimented with

different critical operand table capacities found that a table with 8192 entries performed as well as tables with larger capacities.

We evaluated the effectiveness of the critical operand steering policy by simulating benchmarks executing on a 16-wide, 4-cluster machine described in Section 3.3. We simulated configurations with perfect and Alpha 21264-like branch prediction. In addition, these simulations used the CRF mechanism with a 1 cycle detect-to-set latency, described in Section 5.2.1, to orchestrate inter-cluster communication.

For the experiments with perfect branch prediction we found that the critical-operand steering policy performs marginally better than the dependence and load-slice mechanisms. On the other hand, the mod3 mechanism performs better than critical-operand steering. In the perfect prediction experiments the pipeline is always full of useful instructions. Therefore, a large fraction of the cycles spent waiting for remote operands can be overlapped with the execution of other instructions in the pipeline. In this situation, the mod3 policy performs better than the other three policies because it makes best use of the processor's issue window capacity and issue-width and therefore has greater opportunity to overlap remote operand latencies with useful work.

We also examined the IPC improvement that can be obtained from last-cluster based memory steering by simulating configurations with Alpha 21264-like branch and memory dependence predictors. Figure 6.7 shows the IPC of an 16-wide, 4-cluster machine with the different steering policies nor-

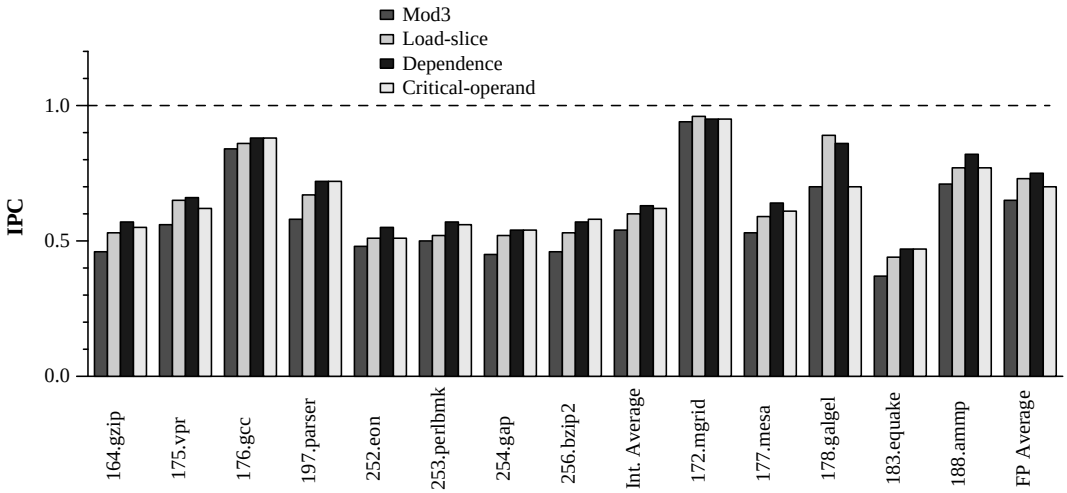

Figure 6.7: The IPC of a 16-wide 4-cluster processor using critical-operand steering. All configurations used Alpha 21264-like branch prediction and memory dependence prediction. Also, the configurations in these experiments used the CRF method to remove transfer instructions.

malized by the IPC of a monolithic machine. Critical-operand steering improves IPC on average by 16% for integer benchmarks and 10% for floating point benchmarks over mod3 steering. However, the dependence and load-slice mechanisms outperform critical-operand steering.

The critical-operand steering policy steers instructions to the same cluster as the instruction that produces their critical source operand. It shows IPC improvement when compared to the mod3 steering policy because it requires fewer remote register operands. Though this policy produces a better instruction distribution, and therefore fewer resource conflicts, compared to dependence and load-slice steering it requires a greater number of remote operands compared to both those mechanisms. Therefore, both dependence and load-

slice steering perform better than critical-operand steering.

6.3 Issue-width Balance Steering

In this section we propose and evaluate the *issue-width balance* steering policy that attempts to improve performance by reducing the number of issue-limited and structure capacity stall cycles. This steering policy keeps track of the number of independent instructions available in each cluster and uses this information to avoid overloading clusters. This steering mechanism works as follows. When an instruction (I1) reaches the steer stage this policy attempts to steer I1 the same cluster as the producers of its input operands (e.g. Cluster 0), just like in dependence steering. However, if Cluster 0 has more ready instructions than it can issue, I1 is assigned to the cluster with the fewest number of independent instructions.

This steering policy attempts to place dependent instructions in the same cluster and thereby avoid the inter-cluster communication penalty and at the same time it also makes better utilization of processor issue width. To quantify the effectiveness of the issue-balance steering policy we evaluated the performance of SPEC 2000 benchmarks executing on a clustered machine described in Section 3.3. We simulated configurations with Alpha 21264-like branch prediction and perfect branch prediction. In addition, these simulations used the CRF mechanism with a 1 cycle detect-to-set latency, described in Section 5.2.1, to orchestrate inter-cluster communication.

Figure 6.8 shows the IPC of an 16-wide, 4-cluster machine with the

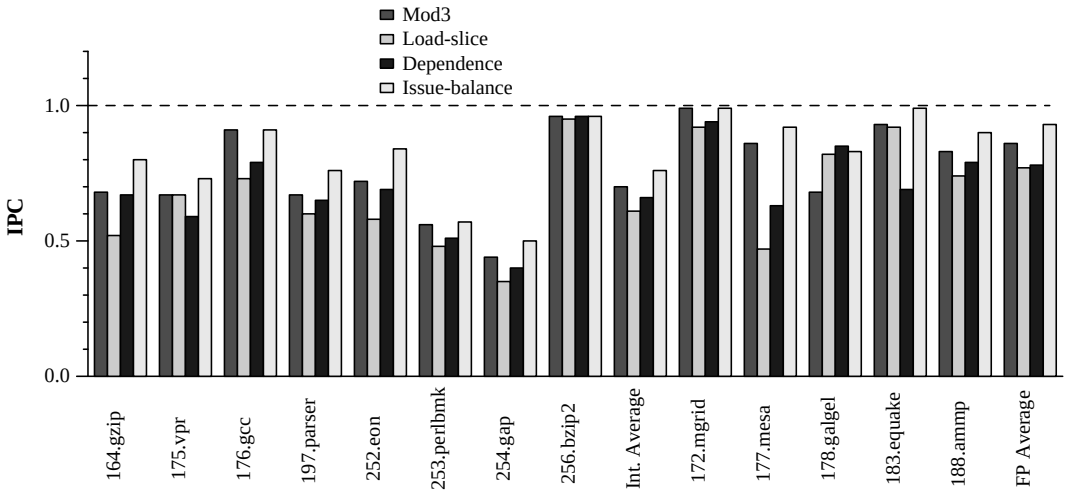

Figure 6.8: The IPC of a 16-wide 4-cluster processor using issue-balance steering. All configurations simulated perfect branch prediction and memory disambiguation. Also, the configurations in these experiments used the CRF method to remove transfer instructions.

different steering policies normalized by the IPC of a monolithic machine. All configurations shown in this figure simulated perfect branch prediction and perfect memory disambiguation. The issue-balance steering policy performs better than the mod3 policy because it requires fewer remote operands. Issue-balance steering also performs better than dependence and load-slice policies because it reduces the number of issue-limited cycles compared to these steering mechanisms.

We also examined the IPC improvement that can be obtained using issue-balance steering by simulating configurations with Alpha 21264-like branch and memory dependence predictors. Figure 6.9 shows the IPC of an 16-wide, 4-cluster machine with the different steering policies normalized by

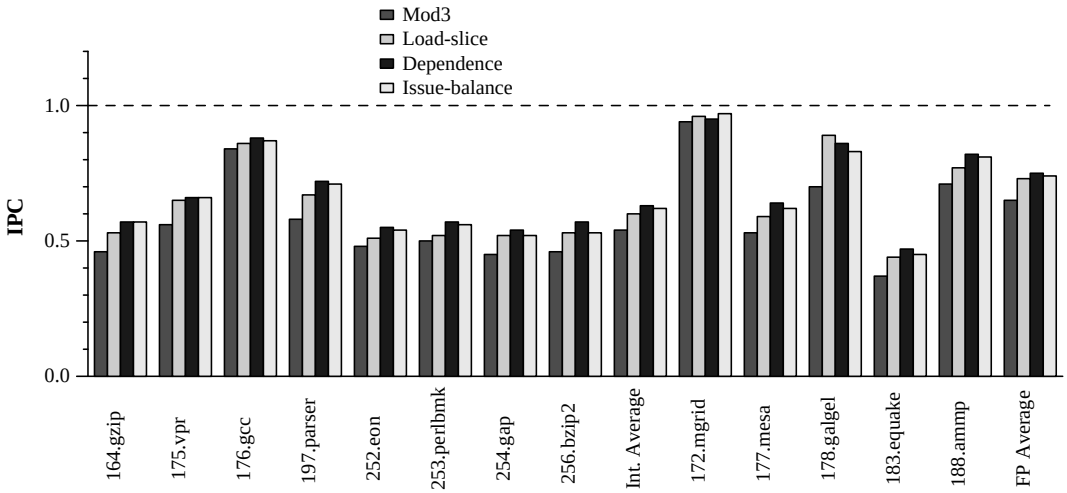

Figure 6.9: The IPC of a 16-wide 4-cluster processor using issue-balance steering. All configurations used Alpha 21264-like branch prediction and memory dependence prediction. Also, the configurations in these experiments used the CRF method to remove transfer instructions.

the IPC of a monolithic machine. The issue-balance steering policy performs better than the mod3 and load-slice policies for all benchmarks except 178.galgel. However, in spite of reducing the number of issue-limited instructions and structure capacity stalls it does not perform better than dependence steering. In the experiments simulating Alpha 21264-like branch prediction the issue window is not always full with useful instructions and many of the structure capacity stalls and issue-limited cycles are caused by instructions that are in the wrong path. Reducing the stalls caused by mis-predicted instructions does not improve performance.

6.4 Related Work

Instruction steering mechanisms for clustered processors is an active area of research within computer architecture. Several steering methods have been proposed to dynamically steer instructions to clusters. The dependence steering policy, that we use as a baseline in several experiments, was proposed by Palacharla *et al.* [50]. This steering policy steers instructions to the same cluster as the producer of one of their source operands. If the producers are in two different clusters we steer the consumer instruction to the producer cluster with the fewest instructions.

Bhargava and John proposed a technique to steer instructions for clustered processors with trace caches [6, 7]. In this technique, like in dependence steering, instructions are assigned to the same cluster as their producers. However, this steering is done at retire time thereby reducing issue-time complexity.

Fields *et al.* proposed a technique to dynamically identify critical paths in a program [23, 24]. They use this method to steer all instructions in the critical path to the same cluster to avoid inter-cluster communication along this path. The critical-operand steering method that we proposed treats does not differentiate between critical-path instructions and other instructions. It attempts to reduce the execution latency of all instructions. The critical-path steering policy specifically identifies critical instructions and schedules them in a manner that reduces their execution latency. Therefore, critical-path steering will perform better than the critical-operand method.

Canal *et al.* proposed the load-store-slice (LdSt slice) steering policy. This technique attempts to steer all instructions that are part of the address computation of a load or store instruction to the same cluster [14]. In their paper, the authors describe the hardware support required to dynamically identify the "backward slice" of load and store instructions. The authors show that the dynamic LdSt-slice policy that they propose achieves a speedup of 16% over a static instruction steering policy, previously suggested by Sastry *et al.* [58]. They also examined a similar steering policy that steers the backward slice of a branch instruction to the same cluster and found that LdSt-slice steering performs better than the Branch-slice policy.

The LdSt-slice steering results in an poor workload balance between clusters and to reduce this imbalance Canal *et al.* proposed a balancing mechanism. They use two parameters to identify an workload imbalance between clusters—the total number of instructions and the number of ready instructions in each cluster. Based on these two parameters they compute a load-balance metric and if this metric is above a pre-determined threshold value they assume that the current instruction distribution has resulted in a workload imbalance. The details of how these parameters are used to compute the load-balance metric is described in [14]. When an imbalance is detected instructions are steered to the cluster with the lighter load in order to re-balance the workload. Though the balancing mechanism reduces workload imbalance it increases the amount of communication between clusters and therefore does not improve the performance of the LdSt-slice steering policy. The technique

described above can be used to balance the workload between two clusters and cannot me extended to three or greater number of clusters. On the other hand, the issue-balance steering policy that we propose can be used to schedule instructions in a processor with greater that two clusters.

Parcerisa *et al.* proposed a steering policy that attempts to reduce the amount of inter-cluster communication and also to balance instruction workload [51]. In their mechanism instructions are assigned to the same cluster as one of their parent instructions unless the workload imbalance between clusters is greater than a given threshold.

Baniasadi and Moshovos proposed and evaluated a number of steering policies for clustered superscalar processors [5]. Among the policies they proposed, the "voting-based" policy is closest to the steering mechanisms that we discuss in this chapter. This steering method attempts to reduce the number of stalls due to limited per-cluster issue-width. The voting-based steering policy starts with an underlying non-adaptive technique and attempts to identify instruction distributions that cause problems. This method uses a cluster prediction table (CPT) that is indexed by the instruction's program counter. Each entry in the table has four 2-bit saturating counters. This table is used to record the instances when an instruction could be issued as soon as its operands are ready. For example, if an instruction, steered to cluster 1, was issued as soon as both its source operands were ready the counter corresponding to cluster 1 is incremented. If the instruction was not issued immediately after its operands are ready the corresponding counter is decremented. Instructions are

assigned to the cluster with the highest CPT counter value. In the event of a tie the instruction is assigned based on the under-lying non-adaptive policy.

Yoaz *et al.* proposed using prediction mechanisms based on bank access history [76] to steer memory instructions. They evaluated several binary predictors that capture the cache-bank access pattern for memory instructions. In addition, they also use a confidence mechanism to determine if the cache-bank for a memory instruction should be predicted. They report that binary predictors can can predict the cache bank that will be accessed by memory instructions with an accuracy of 97%. However, the confidence mechanism filters the instructions with low confidence. Predictions were made for only 50%-70% of memory instructions. Instructions with low prediction-confidence are *replicated* on both clusters. Therefore, their unfiltered cache-bank prediction accuracies range between 49% to 68%.

The last-cluster mechanism, that we proposed, has a prediction accuracy (55%) comparable to the binary predictor's filtered accuracy. Also, the last-cluster mechanism does not require memory instructions to be replicated. Though binary predictors provide slightly better cache-bank prediction compared to last-cluster prediction, they have greater design complexity and could also result in greater power consumption. For these reasons, the last-cluster prediction method would be a better design choice than the binary predictor based mechanism.

Racunas and Patt proposed a method to reduce cache access latency for a partitioned cache [56]. In their technique cache lines could be mapped

to either of two level-1 cache banks and so they have two ways to reduce the cache access latencies—directing memory instructions to the correct cluster or moving cache lines between cache banks. They use two partition assignment tables (PAT) to steer instructions and to track the level-1 bank that a cache line has been mapped to. Instructions are assigned to clusters corresponding to their partition identifiers.

Agarwal proposed a hybrid steering mechanism that selects between memory steering (with last-cluster prediction) and dependence steering [1]. This hybrid mechanism uses 2-bit confidence counters to determine the confidence in the last-cluster prediction. Memory instructions are steered based on the last-cluster table only if the confidence in the prediction is high. If the confidence is low, the instruction is steered based on dependence prediction. Also, in their processor model they allowed cache lines to be dynamically mapped to the level-1 cache banks.

Both PAT steering and hybrid steering allow cache lines to be dynamically re-mapped to any cache bank. Such dynamic cache-line mapping may allow these methods to perform better than the memory-steering policy that we proposed. However, dynamic mapping of cache lines will require additional buses between the level-1 cache banks and the level-2 cache. It will also increase the overall power consumption of the processor.

6.5 Summary

The dependence steering policy, by optimizing for inter-cluster communication alone, suffers an inordinate number of stall cycles due to poor processor resource utilization. The mod3 policy attempts to fully utilize the processor issue-window but in the bargain it generates an excessive amount of inter-cluster communication. We examined three new steering policies that attempt to address the aforementioned problems with dependence and mod3 steering—issue-balance steering, memory steering, and critical-operand steering.

The issue-balance steering policy attempts to steer dependent instructions to the same cluster to reduce inter-cluster communication. However, if such an assignment will result in a cluster having more ready instructions than it can issue, instructions are re-assigned to another cluster to avoid stalls due to limited cluster issue-bandwidth. We found that the issue-balance policy performs better than mod3 and load-slice steering for most benchmarks. However, it offers no improvement over dependence steering. While issue-balance steering reduces the number of *issue-limited* stall cycles it requires more inter-cluster communication compared to dependence steering. The benefit from reducing issue-limited cycles is offset by the additional communication required. In the future, as branch prediction techniques improve, limited per-cluster issue-bandwidth will become a more significant problem. In such a scenario issue-balance steering will be a more effective policy compared to dependence steering.

The critical-operand steering policy identifies which of the two source operands for an instruction is more critical and steers the instruction to the cluster that has fast access to that source operand. This policy performs better than mod3 and load-slice steering for most benchmarks. However, on average, it does not perform better than the dependence steering mechanism.

Memory steering works in conjunction with the baseline policies. This policy steers memory instructions to the cluster that will have local access to their data. Non-memory instructions are steered based on the baseline steering policy (i.e. mod3, load-slice or dependence). Memory steering improves IPC by 5-12% across benchmarks.

Of the three steering mechanisms that we proposed in this chapter we found that critical-operand steering does not improve IPC compared to the baseline mechanisms. Issue-balance steering outperforms the baseline policies in the experiments with perfect branch prediction. For the experiments with an Alpha-21264 like branch predictor we found that this policy does only as well as dependence steering. We found that the memory steering policy, used in conjunction with dependence steering, provides the best performance among the three policies that we proposed. The effectiveness of this policy can be further improved by increasing the cache bank prediction accuracy for memory instructions.

Chapter 7

Conclusions

Historically processor performance has been improved by increasing clock frequency and IPC. Increasing the processor pipeline depth (fewer gates per cycle) has been one technique that has been used to improve clock frequency. In this research we examined how much further reducing the amount of logic per pipeline stage can improve performance.

Increasing processor pipeline depth improves clock frequency. However, there are certain critical sections of the pipeline, termed critical loops, that must operate in the fewest possible cycles to achieve good performance. Increasing pipeline depth increases the latency of these critical loops and in turn reduces IPC. Thus, there is a tradeoff between increasing pipeline depth (clock frequency) and IPC. To obtain maximum performance processor designers must balance pipeline depth and IPC. We explored this tradeoff for an Alpha 21264 processor. Increasing pipeline depths improves performance up till the point where the reduction in IPC outweighs the benefits of increased clock frequency. Further increasing the pipeline depth beyond this optimal point will degrade overall performance. We determined that the amount of useful logic per stage that will provide the best performance is approximately

6 FO4 inverter delays.

Our pipeline scaling study shows that clock improvements from processor pipelines are approaching a point diminishing return. Novel architectural techniques targeted at reducing the effect of critical loops may allow designers to increase pipeline depths beyond the 6 FO4 optimal point. However, critical loops in modern processors, such as the Pentium IV, are already aggressively pipelined and are of significant design complexity. It is unlikely that structures that are part of critical loops can be pipelined much further.

Microprocessor performance has improved at about 55% per year for the last three decades, with much of the gains resulting from higher clock frequencies, due to process technology and deeper pipelines. However, our results show that pipelining can contribute *at most* another factor of two to clock rate improvements. Subsequently, in the best case, clock rates will increase at the rate of feature size scaling, which is projected to be 12-20% per year. Any additional performance improvements must come from increases in concurrency, whether they be instruction-level parallelism, thread-level parallelism, or a combination of the two.

To exploit greater parallelism in the instruction stream processors will have to issue more instructions every cycle. In addition, the capacity and the number of ports of on-chip structures such as the register file, the instruction issue-window, and the data-cache must also be increased. However, large multi-ported structures will have long access latencies that will not scale with technology. Such structures cannot be clocked at aggressive frequencies. A

natural solution to the problem of increasing circuit complexity is to partition the architecture into clusters. Each of the processors on-chip structures is divided among the clusters and therefore the complexity of each individual piece is reduced. Farkas *et al.* proposed a clustered superscalar architecture [22] called the Multicluster architecture. We use this architecture as our baseline.

While clustering reduces the complexity of on-chip structures it introduces other bottlenecks and inefficiencies. The three primary bottlenecks in clustered processors are–(1) inter-cluster communication delays, (2) transfer instruction overhead, and (3) cluster resource limitation. These bottlenecks reduce the IPC of a clustered machine compared to an ideal monolithic machine. We found that clustering lowered IPC between 29-43% compared to an ideal unclustered machine. In this research we proposed techniques to reduce the effect of the bottlenecks from clustering. Our goal is to improve the IPC of a clustered processor to be closer to that of the ideal monolithic machine.

7.1 Dissertation Summary

We proposed and evaluated two mechanisms that replace transfer instructions with hardware signals—(a) consumer requested forwarding (CRF) and (b) hot-register based forwarding. In the CRF method, consumer instructions that require values from remote clusters explicitly request the value to be forwarded by setting a bit in a structure called the inter-cluster forward bit (IFB) table. Producer instructions, upon completion of execution, consult

this table to determine if the value should be forwarded to other clusters.

Inter-cluster dependence between producer and consumer instructions is detected when the consumer instruction reaches the rename stage of the pipeline. When such a dependence is detected the corresponding forward bit it set in the producer cluster's IFB table. The time between detecting the dependence to setting the forward-bit is termed the detect-to-set delay. We found that the benefits of the CRF policy are reduced as the detect-to-set delay increases. For detect-to-set latencies of 3 cycles and greater the CRF mechanism provides no performance improvement.

The hot-registers mechanism tracks the registers that are used by each cluster and uses this information to predict where instruction outputs should be forwarded. These predictions are made at the time the producer instruction reaches the rename stage of the pipeline. Though the hot-register forwarding mechanism shows improvement over the baseline this improvement is lower than the CRF method with a 1-cycle detect-to-set delay.

The two mechanisms that we proposed to orchestrate inter-cluster communication remove almost all transfer instructions. The IPC improvement from these methods range between 1% to 9% over the baseline processor. Such small gains in performance is not worth the additional design complexity that these mechanisms will introduce. Transfer instructions will become a significant bottleneck only if the pipeline is frequently full with useful instructions. Therefore, these techniques will be useful only if researchers improve control and memory prediction significantly.

In this research we also proposed steering mechanisms to reduce the inter-cluster communication bottleneck and the per-cluster resource bottleneck. We proposed and examined three new steering policies—issue-balance steering, memory steering, and critical-operand steering.

The issue-balance steering policy attempts to steer dependent instructions to the same cluster to reduce inter-cluster communication. However, if such an assignment will result in a cluster having more ready instructions than it can issue, instructions are re-assigned to another cluster to avoid stalls due to limited cluster issue-bandwidth. We found that the issue-balance policy performs better than two of the baseline steering policies—mod3 and load-slice steering—for most benchmarks. However, it performs only as well as the dependence steering.

The critical-operand steering policy identifies which of the two source operands for an instruction is more critical and steers the instruction to the cluster that has fast access to that source operand. This policy also performs better than mod3 and load-slice steering for most benchmarks. However, on average, it does not perform better than the dependence steering mechanism.

Memory-steering works in conjunction with the baseline policies. This policy steers memory instructions to the cluster that will have local access to their data. Non-memory instructions are steered based on the baseline steering policy. Memory-steering improves IPC by 5-12% across benchmarks.

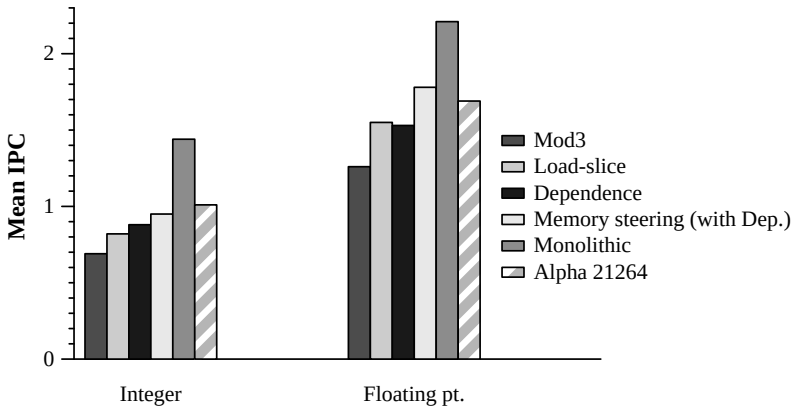

Figure 7.1: The IPCs of a 16-wide clustered processor, a 16-wide monolithic processor, and an Alpha 21264-like configuration. All these simulations used a tournament style predictor like in the Alpha-21264.

7.2 Discussion

This research proposed and evaluated several techniques to reduce the effect of bottlenecks in clustered processors and improve their IPC. Figure 7.1 shows the IPC of the baseline clustered processor, a clustered processor that uses the CRF policy and our best steering mechanism (memory steering in conjunction with dependence steering), the IPC of a monolithic 16-wide processor, and a Alpha 21264-like (6-wide) configuration. All these experiments used a tournament style branch predictor like in the Alpha 21264.

For integer benchmarks the IPC of a 16-wide monolithic machine is 43% greater than an Alpha 21264 (6-wide). For floating-point benchmarks

the IPC of a 16-wide monolithic machine is 31% greater than an Alpha 21264. Even though the issue width of the 16-wide monolithic processor is more than double that of the Alpha 21264, the improvements in IPC are at best 43%. It will not be worth increasing the area of the execution core (almost double) and increasing processor power for such a small increase in IPC.

Furthermore, the issue width of the Alpha 21264 is close to that of each cluster in the 16-wide clustered processor. Therefore, it is reasonable to assume that a 6-wide Alpha 21264 can be clocked at the same frequency as the 16-wide clustered processor. The IPC of the Alpha 21264 is 6% greater than the best clustered processor configuration for integer benchmarks. For floating-point benchmarks the IPC of the Alpha 21264 is 5% lower than the IPC of the best clustered processor configuration.

Instruction supply is the primary bottleneck in wide-issue processors. This bottleneck is the reason why the 16-wide clustered processor does not show significant improvement over the Alpha 21264. It will unviable to design wide-issue processors unless researchers improve branch prediction accuracies. In our studies we also examined configurations that simulated perfect branch prediction. Figure 7.2 shows the average IPC of a 16-wide clustered processor and an Alpha 21264 processor for configurations with perfect branch prediction. Using the techniques that we proposed improves the IPC of the clustered processor significantly. For these experiments the IPC of a 16-wide clustered processor (best case) is 53% greater than an Alpha 21264 processor for integer benchmarks (41% for floating pt.). The clustered processor with CRF based

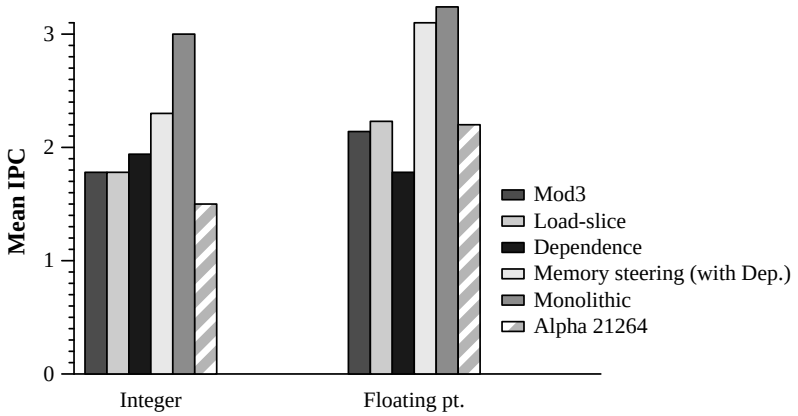

Figure 7.2: The IPCs of a 16-wide clustered processor, a 16-wide monolithic processor, and an Alpha 21264-like configuration. All these simulations used a perfect branch prediction.

forwarding and memory steering has an IPC that is 23% lower than a monolithic machine for integer benchmarks. For floating point benchmarks the IPC of the clustered processor is only 5% lower.

The techniques presented in this dissertation solve most of the bottlenecks in the execution core of clustered processors. Several architectural ideas have already been proposed to improve branch prediction [16, 28, 37, 52, 57, 60] and future research will provide further improvements. The architectural ideas proposed in this research will acquire greater significance as instruction supply techniques improve.

Appendices

Appendix A

Pipeline Scaling Simulation Results

Benchmark	ϕ_{logic}						
	2	3	4	5	6	7	8
164.gzip	0.22	0.32	0.39	0.51	0.65	0.66	0.70
175.vpr	0.21	0.31	0.38	0.47	0.59	0.60	0.64
176.gcc	0.18	0.26	0.32	0.38	0.45	0.47	0.50
181.mcf	0.18	0.28	0.33	0.42	0.50	0.53	0.55
197.parser	0.18	0.26	0.32	0.39	0.48	0.48	0.52
252.eon	0.19	0.27	0.34	0.43	0.51	0.55	0.60
253.perlbmk	0.16	0.22	0.27	0.33	0.40	0.40	0.43
256.bzip2	0.27	0.38	0.48	0.55	0.67	0.68	0.71
171.swim	0.49	0.63	0.77	0.80	0.83	0.83	0.83
172.mgrid	0.37	0.52	0.68	0.83	0.95	1.04	1.13
173.applu	0.26	0.36	0.46	0.54	0.60	0.64	0.67
177.mesa	0.23	0.33	0.42	0.52	0.62	0.67	0.73
178.galgel	0.26	0.37	0.48	0.60	0.68	0.79	0.85
179.art	0.17	0.23	0.28	0.31	0.34	0.35	0.37
183.equake	0.20	0.30	0.37	0.47	0.59	0.58	0.64
188.ammp	0.05	0.07	0.09	0.10	0.11	0.11	0.12
189.lucas	0.28	0.41	0.53	0.67	0.78	0.93	0.99

Table A.1: The IPCs of SPEC 2000 benchmarks at pipeline depths corresponding to ϕ_{logic} between 2 and 8 FO4

Benchmark	ϕ_{logic}							
	9	10	11	12	13	14	15	16
164.gzip	0.88	0.95	0.97	0.98	0.98	0.98	1.04	1.04
175.vpr	0.79	0.86	0.88	0.88	0.88	0.89	0.97	0.97
176.gcc	0.57	0.63	0.65	0.65	0.66	0.66	0.69	0.69
181.mcf	0.65	0.72	0.75	0.74	0.75	0.75	0.78	0.78
197.parser	0.62	0.67	0.69	0.70	0.70	0.70	0.73	0.73
252.eon	0.68	0.76	0.78	0.83	0.84	0.88	0.93	0.93
253.perlbmk	0.50	0.55	0.57	0.57	0.57	0.58	0.62	0.62
256.bzip2	0.83	0.91	0.96	0.96	0.96	0.96	1.01	1.01
171.swim	0.83	0.84	0.84	0.84	0.84	0.85	0.85	0.88
172.mgrid	1.22	1.30	1.32	1.38	1.39	1.43	1.45	1.45
173.applu	0.71	0.76	0.74	0.79	0.79	0.81	0.82	0.82
177.mesa	0.83	0.93	0.93	0.98	0.99	1.04	1.09	1.08
178.galgel	0.94	1.04	1.03	1.15	1.15	1.28	1.29	1.28
179.art	0.38	0.40	0.41	0.42	0.42	0.42	0.42	0.42
183.equake	0.75	0.82	0.85	0.86	0.86	0.86	0.90	0.90
188.ammp	0.13	0.14	0.14	0.15	0.15	0.15	0.16	0.16
189.lucas	1.10	1.22	1.25	1.36	1.36	1.51	1.51	1.51

Table A.2: The IPCs of SPEC 2000 benchmarks at pipeline depths corresponding to ϕ_{logic} between 9 and 16 FO4

Appendix B

Clustered Processor Simulation Results

B.1 The Baseline Clustered Processor

Benchmark	Monolithic	Mod3	Load-slice	Dependence
164.gzip	3.47	1.77	1.77	2.16
175.vpr	1.98	0.96	1.22	1.17
176.gcc	1.83	1.48	1.35	1.44
181.mcf	0.11	0.11	0.11	0.11
197.parser	2.65	1.37	1.54	1.71
252.eon	4.29	2.31	2.50	2.92
253.perlbmk	2.03	0.98	0.99	1.04
254.gap	3.77	1.17	1.31	1.43
256.bzip2	3.75	3.42	3.55	3.59
171.swim	0.60	0.60	0.60	0.60
172.mgrid	1.28	1.23	1.18	1.19
173.applu	0.62	0.59	0.57	0.59
177.mesa	4.85	2.72	2.28	2.89
178.galgel	4.13	2.44	3.11	3.29
179.art	0.38	0.38	0.37	0.38
183.equake	3.92	2.88	3.15	2.54
188.ammp	2.01	1.43	1.50	1.54
189.lucas	0.40	0.40	0.40	0.40

Table B.1: The IPCs of a monolithic and a 16-wide 4-cluster processor. These configurations simulated perfect branch prediction and perfect memory disambiguation.

Benchmark	Monolithic	Mod3	Load-slice	Dependence
164.gzip	1.79	0.73	0.92	0.99
175.vpr	0.93	0.47	0.60	0.61
176.gcc	1.13	0.90	0.95	0.99
181.mcf	0.11	0.11	0.11	0.11
197.parser	1.36	0.73	0.91	0.96
252.eon	1.38	0.62	0.67	0.75
253.perlbmk	0.93	0.44	0.47	0.53
254.gap	1.28	0.53	0.63	0.66
256.bzip2	2.65	1.08	1.35	1.49
171.swim	0.60	0.60	0.60	0.60
172.mgrid	1.25	1.15	1.19	1.17
173.applu	0.59	0.55	0.55	0.56
177.mesa	1.98	0.97	1.12	1.24
178.galgel	3.74	2.31	3.25	2.90
179.art	0.35	0.34	0.35	0.35
183.equake	2.59	0.90	1.09	1.18
188.ammp	1.44	0.93	1.09	1.15
189.lucas	0.40	0.40	0.40	0.40

Table B.2: The IPCs of a monolithic and a 16-wide 4-cluster processor. These configurations simulated Alpha 21264-like branch prediction and memory dependence prediction.

Benchmark	Monolithic	Mod3	Load-slice	Dependence
164.gzip	3.24	1.35	1.03	1.31
175.vpr	1.95	0.88	1.06	0.99
176.gcc	1.79	1.14	0.90	1.02
181.mcf	0.11	0.11	0.10	0.10
197.parser	2.52	1.17	1.05	1.25
252.eon	4.29	1.75	1.36	1.71
253.perlbmk	1.86	0.85	0.73	0.79
254.gap	3.41	1.09	0.96	1.23
256.bzip2	3.67	2.24	2.58	2.83
171.swim	0.60	0.60	0.60	0.60
172.mgrid	1.27	1.20	1.10	1.11
173.applu	0.61	0.57	0.52	0.56
177.mesa	4.58	1.77	1.18	1.52
178.galgel	4.13	2.43	2.59	3.02
179.art	0.38	0.37	0.36	0.37
183.equake	3.92	1.90	1.66	1.28
188.ammp	1.97	1.20	1.00	1.04
189.lucas	0.40	0.40	0.40	0.40

Table B.3: The IPCs of a monolithic and a 8-wide 4-cluster processor. These configurations simulated perfect branch prediction and perfect memory disambiguation.

Benchmark	Monolithic	Mod3	Load-slice	Dependence
164.gzip	5.12	1.94	2.32	2.66
175.vpr	2.05	1.01	1.29	1.25
176.gcc	1.98	1.55	1.54	1.61
181.mcf	0.11	0.11	0.11	0.11
197.parser	2.98	1.42	1.70	1.86
252.eon	6.48	2.51	3.33	3.67
253.perlbmk	2.26	1.02	1.12	1.16
254.gap	4.02	1.20	1.41	1.48
256.bzip2	6.67	4.09	4.79	5.95
171.swim	0.60	0.60	0.60	0.60
172.mgrid	1.28	1.24	1.19	1.21
173.applu	0.63	0.60	0.60	0.61
177.mesa	8.13	3.13	3.39	3.96
178.galgel	4.13	2.44	3.02	3.27
179.art	0.38	0.38	0.38	0.38
183.equake	8.49	3.42	4.58	4.27
188.ammp	2.48	1.54	1.86	1.94
189.lucas	0.40	0.40	0.40	0.40

Table B.4: The IPCs of a monolithic and a 32-wide 4-cluster processor. These configurations simulated perfect branch prediction and perfect memory disambiguation.

Benchmark	Mod3	Load-slice	Dependence
164.gzip	2.39	1.80	2.34
175.vpr	1.38	1.33	1.19
176.gcc	1.69	1.36	1.45
181.mcf	0.11	0.11	0.11
197.parser	1.84	1.59	1.75
252.eon	3.17	2.50	2.98
253.perlbmk	1.16	0.99	1.05
254.gap	1.75	1.33	1.53
256.bzip2	3.61	3.60	3.62
171.swim	0.60	0.60	0.60
172.mgrid	1.27	1.19	1.20
173.applu	0.62	0.57	0.60
177.mesa	4.35	2.29	3.11
178.galgel	2.88	3.54	3.49
179.art	0.38	0.37	0.38
183.equake	3.74	3.63	2.74
188.ammp	1.70	1.53	1.60
189.lucas	0.40	0.40	0.40

Table B.5: The IPC of an 16-wide 4-cluster processor with no transfer instructions. These configurations simulated perfect branch prediction and perfect memory disambiguation.

Benchmark	Mod3	Load-slice	Dependence
164.gzip	0.93	0.98	1.03
175.vpr	0.56	0.61	0.63
176.gcc	0.96	1.00	1.00
181.mcf	0.11	0.11	0.11
197.parser	0.86	0.95	1.00
252.eon	0.74	0.71	0.77
253.perlbmk	0.52	0.50	0.54
254.gap	0.63	0.67	0.70
256.bzip2	1.32	1.43	1.53
171.swim	0.60	0.60	0.60
172.mgrid	1.20	1.21	1.20
173.applu	0.55	0.56	0.56
177.mesa	1.15	1.21	1.30
178.galgel	2.65	3.34	3.18
179.art	0.35	0.35	0.35
183.equake	1.20	1.17	1.24
188.ammp	1.05	1.14	1.23
189.lucas	0.40	0.40	0.40

Table B.6: The IPC of an 16-wide 4-cluster processor with no transfer instructions. These configurations simulated Alpha 21264-like branch prediction and memory dependence prediction.

Benchmark	Mod3	Load-slice	Dependence
164.gzip	2.18	1.06	1.51
175.vpr	1.35	1.18	1.03
176.gcc	1.65	0.90	1.05
181.mcf	0.11	0.11	0.11
197.parser	1.77	1.12	1.33
252.eon	3.04	1.36	1.77
253.perlbmk	1.10	0.73	0.81
254.gap	1.69	0.98	1.33
256.bzip2	3.60	3.03	3.36
171.swim	0.60	0.60	0.60
172.mgrid	1.27	1.11	1.15
173.applu	0.61	0.52	0.57
177.mesa	3.87	1.19	1.75
178.galgel	2.88	3.40	3.46
179.art	0.38	0.36	0.37
183.equake	3.40	2.11	1.40
188.ammp	1.66	1.05	1.12
189.lucas	0.40	0.40	0.40

Table B.7: The IPC of an 8-wide 4-cluster processor with no transfer instructions. These configurations simulated perfect branch prediction and perfect memory disambiguation.

Benchmark	Mod3	Load-slice	Dependence
164.gzip	2.53	2.33	2.71
175.vpr	1.41	1.40	1.26
176.gcc	1.76	1.54	1.61
181.mcf	0.11	0.11	0.11
197.parser	1.91	1.74	1.88
252.eon	3.44	3.33	3.70
253.perlbmk	1.19	1.12	1.16
254.gap	1.78	1.42	1.58
256.bzip2	4.69	4.85	5.97
171.swim	0.60	0.60	0.60
172.mgrid	1.28	1.19	1.21
173.applu	0.62	0.60	0.61
177.mesa	4.83	3.39	4.06
178.galgel	2.88	3.38	3.40
179.art	0.38	0.38	0.38
183.equake	5.19	5.30	4.40
188.ammp	1.84	1.89	1.99
189.lucas	0.40	0.40	0.40

Table B.8: The IPC of an 32-wide 4-cluster processor with no transfer instructions. These configurations simulated perfect branch prediction and perfect memory disambiguation.

Benchmark	Mod3	Load-slice	Dependence
164.gzip	2.54	1.95	2.37
175.vpr	1.31	1.46	1.33
176.gcc	1.62	1.37	1.47
181.mcf	0.11	0.11	0.11
197.parser	1.81	1.65	1.83
252.eon	3.18	2.62	3.12
253.perlbmk	1.33	1.16	1.23
254.gap	2.18	1.87	2.34
256.bzip2	3.58	3.57	3.62
171.swim	0.60	0.60	0.60
172.mgrid	1.24	1.19	1.19
173.applu	0.59	0.57	0.59
177.mesa	3.34	2.32	3.01
178.galgel	3.32	3.70	3.77
179.art	0.38	0.37	0.38
183.equake	3.71	3.30	2.56
188.ammp	1.78	1.54	1.60
189.lucas	0.40	0.40	0.40

Table B.9: The IPC of an 16-wide 4-cluster processor without the inter-cluster communication bottleneck. These configurations simulated perfect branch prediction and perfect memory disambiguation.

Benchmark	Mod3	Load-slice	Dependence
164.gzip	1.39	1.33	1.39
175.vpr	0.75	0.76	0.76
176.gcc	1.06	1.07	1.07
181.mcf	0.10	0.10	0.10
197.parser	1.12	1.13	1.15
252.eon	1.12	1.05	1.10
253.perlbmk	0.74	0.68	0.71
254.gap	1.09	1.10	1.12
256.bzip2	2.22	2.40	2.47
171.swim	0.60	0.60	0.60
172.mgrid	1.21	1.21	1.20
173.applu	0.57	0.57	0.57
177.mesa	1.64	1.62	1.72
178.galgel	3.26	3.57	3.44
179.art	0.35	0.35	0.35
183.equake	1.91	1.99	2.14
188.ammp	1.29	1.32	1.35
189.lucas	0.40	0.40	0.40

Table B.10: The IPC of an 16-wide 4-cluster processor without the inter-cluster communication bottleneck. These configurations simulated Alpha 21264-like branch prediction and memory dependence prediction.

Benchmark	Mod3	Load-slice	Dependence
164.gzip	1.62	1.05	1.35
175.vpr	1.15	1.21	1.08
176.gcc	1.30	0.91	1.03
181.mcf	0.11	0.10	0.10
197.parser	1.43	1.07	1.28
252.eon	2.06	1.37	1.74
253.perlbmk	1.07	0.79	0.87
254.gap	1.65	1.02	1.70
256.bzip2	2.29	2.59	2.83
171.swim	0.60	0.60	0.60
172.mgrid	1.22	1.10	1.11
173.applu	0.57	0.52	0.56
177.mesa	1.88	1.19	1.54
178.galgel	3.22	2.62	3.11
179.art	0.37	0.36	0.37
183.equake	2.08	1.68	1.28
188.ammp	1.32	1.01	1.06
189.lucas	0.40	0.39	0.40

Table B.11: The IPC of an 8-wide 4-cluster processor without the inter-cluster communication bottleneck. These configurations simulated perfect branch prediction and perfect memory disambiguation.

Benchmark	Mod3	Load-slice	Dependence
164.gzip	3.20	2.86	3.25
175.vpr	1.42	1.58	1.46
176.gcc	1.73	1.59	1.66
181.mcf	0.11	0.11	0.11
197.parser	1.96	1.92	2.05
252.eon	4.01	4.13	4.67
253.perlbmk	1.46	1.41	1.47
254.gap	2.33	2.38	2.58
256.bzip2	6.26	6.36	6.43
171.swim	0.60	0.60	0.60
172.mgrid	1.26	1.20	1.21
173.applu	0.61	0.60	0.61
177.mesa	4.72	3.78	4.80
178.galgel	3.33	3.65	3.75
179.art	0.38	0.38	0.38
183.equake	5.81	5.47	4.77
188.ammp	2.12	2.06	2.10
189.lucas	0.40	0.40	0.40

Table B.12: The IPC of an 32-wide 4-cluster processor without the inter-cluster communication bottleneck. These configurations simulated perfect branch prediction and perfect memory disambiguation.

Benchmark	Mod3	Load-slice	Dependence
164.gzip	2.08	3.00	3.00
175.vpr	1.19	1.76	1.76
176.gcc	1.48	1.83	1.83
181.mcf	0.11	0.11	0.11
197.parser	1.74	2.46	2.46
252.eon	2.56	4.09	4.09
253.perlbmk	1.51	1.78	1.78
254.gap	1.78	2.36	2.36
256.bzip2	3.43	3.74	3.74
171.swim	0.61	0.61	0.61
172.mgrid	1.23	1.33	1.32
173.applu	0.60	0.58	0.60
177.mesa	2.72	4.10	4.05
178.galgel	2.45	3.11	3.44
179.art	0.38	0.38	0.38
183.equake	2.89	3.31	3.72
188.ammp	1.44	1.92	1.88
189.lucas	0.40	0.40	0.40

Table B.13: The IPC of an 16-wide 4-cluster processor without the cluster resource limitation bottleneck. These configurations simulated perfect branch prediction and perfect memory disambiguation.

Benchmark	Mod3	Load-slice	Dependence
164.gzip	0.77	1.32	1.36
175.vpr	0.51	0.76	0.77
176.gcc	0.92	0.97	1.04
181.mcf	0.11	0.11	0.11
197.parser	0.79	1.11	1.14
252.eon	0.65	0.95	1.01
253.perlbmk	0.50	0.70	0.74
254.gap	0.53	0.77	0.80
256.bzip2	1.11	1.53	1.62
171.swim	0.61	0.61	0.61
172.mgrid	1.15	1.22	1.19
173.applu	0.57	0.56	0.57
177.mesa	1.00	1.39	1.45
178.galgel	2.35	3.35	2.93
179.art	0.35	0.35	0.35
183.equake	0.96	1.62	1.59
188.ammp	0.95	1.18	1.17
189.lucas	0.40	0.40	0.40

Table B.14: The IPC of an 16-wide 4-cluster processor without the cluster resource limitation bottleneck. These configurations simulated Alpha 21264-like branch prediction and memory dependence prediction.

Benchmark	Mod3	Load-slice	Dependence
164.gzip	2.14	3.10	3.01
175.vpr	1.22	1.67	1.83
176.gcc	1.15	1.79	1.76
181.mcf	0.11	0.11	0.11
197.parser	1.65	2.43	2.41
252.eon	2.70	3.89	3.90
253.perlbmk	1.40	1.81	1.77
254.gap	2.25	3.03	2.73
256.bzip2	2.27	3.23	3.14
171.swim	0.61	0.61	0.61
172.mgrid	1.27	1.27	1.25
173.applu	0.57	0.62	0.60
177.mesa	2.48	4.20	4.25
178.galgel	2.91	3.31	3.62
179.art	0.37	0.38	0.38
183.equake	2.25	3.45	3.41
188.ammp	1.49	1.92	1.88
189.lucas	0.40	0.40	0.40

Table B.15: The IPC of an 8-wide 4-cluster processor without the cluster resource limitation bottleneck. These configurations simulated perfect branch prediction and perfect memory disambiguation.

Benchmark	Mod3	Load-slice	Dependence
164.gzip	2.62	3.56	4.38
175.vpr	1.25	1.66	1.85
176.gcc	1.59	1.94	1.94
181.mcf	0.11	0.11	0.11
197.parser	1.97	2.73	2.77
252.eon	3.58	5.69	5.45
253.perlbmk	1.27	1.88	1.76
254.gap	1.61	2.65	2.23
256.bzip2	4.09	5.04	6.18
171.swim	0.61	0.61	0.61
172.mgrid	1.24	1.27	1.27
173.applu	0.61	0.63	0.63
177.mesa	3.75	5.24	6.20
178.galgel	2.82	3.14	3.53
179.art	0.38	0.39	0.39
183.equake	3.74	5.93	6.87
188.ammp	1.61	2.25	2.27
189.lucas	0.40	0.40	0.40

Table B.16: The IPC of an 32-wide 4-cluster processor without the cluster resource limitation bottleneck. These configurations simulated perfect branch prediction and perfect memory disambiguation.

B.2 Register Caching

Benchmark	Mod3	Load-slice	Dependence
164.gzip	1.89	1.78	2.20
175.vpr	1.03	1.24	1.17
176.gcc	1.54	1.36	1.45
181.mcf	0.11	0.11	0.11
197.parser	1.46	1.55	1.71
252.eon	2.54	2.50	2.95
253.perlbmk	1.02	0.99	1.04
254.gap	1.26	1.32	1.44
256.bzip2	3.48	3.56	3.60
171.swim	0.61	0.61	0.61
172.mgrid	1.25	1.19	1.20
173.applu	0.61	0.58	0.60
177.mesa	3.14	2.28	2.92
178.galgel	2.46	3.12	3.35
179.art	0.38	0.38	0.38
183.equake	3.17	3.21	2.55
188.ammp	1.46	1.51	1.56
189.lucas	0.40	0.40	0.40

Table B.17: The IPC of an 16-wide 4-cluster processor with register caching. These configurations simulated perfect branch prediction and perfect memory disambiguation.

Benchmark	Mod3	Load-slice	Dependence
164.gzip	0.74	0.93	1.01
175.vpr	0.48	0.60	0.62
176.gcc	0.92	0.97	1.00
181.mcf	0.11	0.11	0.11
197.parser	0.75	0.91	0.97
252.eon	0.63	0.68	0.76
253.perlbmk	0.45	0.48	0.53
254.gap	0.53	0.63	0.67
256.bzip2	1.09	1.37	1.47
171.swim	0.61	0.61	0.61
172.mgrid	1.15	1.20	1.18
173.applu	0.55	0.56	0.56
177.mesa	0.99	1.13	1.26
178.galgel	2.33	3.32	2.91
179.art	0.35	0.35	0.35
183.equake	0.90	1.09	1.20
188.ammp	0.95	1.09	1.15
189.lucas	0.40	0.40	0.40

Table B.18: The IPC of an 16-wide 4-cluster processor with register caching. These configurations simulated Alpha 21264-like branch prediction and memory dependence prediction.

B.3 Consumer-requested Forwarding

Benchmark	Detect-to-set delay			
	1	2	3	4
164.gzip	2.37	2.16	1.82	1.68
175.vpr	1.33	1.16	0.79	0.77
176.gcc	1.68	1.63	1.53	1.51
181.mcf	0.11	0.11	0.11	0.11
197.parser	1.79	1.48	1.20	1.13
252.eon	3.10	2.48	1.87	1.77
253.perlbmk	1.15	1.09	1.03	1.01
254.gap	1.66	0.99	0.84	0.78
256.bzip2	3.60	3.30	1.81	1.70
171.swim	0.60	0.60	0.60	0.60
172.mgrid	1.27	1.24	1.15	1.02
173.applu	0.62	0.60	0.58	0.56
177.mesa	4.20	3.05	2.08	1.92
178.galgel	2.83	1.83	1.34	1.26
179.art	0.38	0.38	0.37	0.36
183.equake	3.66	3.04	2.65	2.06
188.ammp	1.68	1.59	1.29	1.23
189.lucas	0.40	0.40	0.40	0.40

Table B.19: The IPC of an 16-wide 4-cluster processor with consumer requested forwarding for different detect-to-set latencies. These simulations used the dual-wakeup policy to avoid deadlocks and used mod3 steering. These configurations simulated perfect branch prediction and perfect memory disambiguation.

Benchmark	Detect-to-set delay			
	1	2	3	4
164.gzip	1.80	1.80	1.80	1.80
175.vpr	1.33	1.20	0.97	0.95
176.gcc	1.36	1.36	1.36	1.36
181.mcf	0.11	0.11	0.11	0.11
197.parser	1.59	1.57	1.56	1.54
252.eon	2.50	2.50	2.50	2.50
253.perlbmk	0.99	0.99	0.99	0.99
254.gap	1.33	1.31	1.30	1.29
256.bzip2	3.60	3.56	3.49	3.45
171.swim	0.60	0.60	0.60	0.60
172.mgrid	1.19	1.19	1.18	1.18
173.applu	0.57	0.57	0.57	0.57
177.mesa	2.29	2.29	2.29	2.29
178.galgel	3.41	3.20	2.98	2.78
179.art	0.37	0.37	0.37	0.37
183.equake	3.62	3.60	3.55	3.49
188.ammp	1.50	1.50	1.49	1.49
189.lucas	0.40	0.40	0.40	0.40

Table B.20: The IPC of an 16-wide 4-cluster processor with consumer requested forwarding for different detect-to-set latencies. These simulations used the dual-wakeup policy to avoid deadlocks and used load-slice steering. These configurations simulated perfect branch prediction and perfect memory disambiguation.

Benchmark	Detect-to-set delay			
	1	2	3	4
164.gzip	2.34	2.34	2.33	2.32
175.vpr	1.18	1.15	1.13	1.12
176.gcc	1.45	1.45	1.45	1.44
181.mcf	0.11	0.11	0.11	0.11
197.parser	1.75	1.73	1.70	1.67
252.eon	2.99	2.98	2.98	2.98
253.perlbmk	1.05	1.05	1.05	1.05
254.gap	1.53	1.50	1.33	1.31
256.bzip2	3.62	3.62	3.59	3.52
171.swim	0.60	0.60	0.60	0.60
172.mgrid	1.20	1.20	1.19	1.19
173.applu	0.60	0.60	0.60	0.60
177.mesa	3.09	3.09	3.07	3.06
178.galgel	3.52	3.14	2.53	2.35
179.art	0.38	0.38	0.38	0.38
183.equake	2.74	2.74	2.74	2.74
188.ammp	1.60	1.60	1.60	1.62
189.lucas	0.40	0.40	0.40	0.40

Table B.21: The IPC of an 16-wide 4-cluster processor with consumer requested forwarding for different detect-to-set latencies. These simulations used the dual-wakeup policy to avoid deadlocks and used dependence steering. These configurations simulated perfect branch prediction and perfect memory disambiguation.

Benchmark	Mod3	Load-slice	Dependence
164.gzip	0.83	0.96	1.02
175.vpr	0.52	0.61	0.62
176.gcc	0.95	0.98	1.01
181.mcf	0.10	0.10	0.10
197.parser	0.78	0.89	0.98
252.eon	0.66	0.70	0.76
253.perlbmk	0.47	0.49	0.53
254.gap	0.57	0.66	0.69
256.bzip2	1.23	1.42	1.52
171.swim	0.60	0.60	0.60
172.mgrid	1.18	1.21	1.19
173.applu	0.55	0.56	0.56
177.mesa	1.04	1.18	1.27
178.galgel	2.62	3.34	3.25
179.art	0.35	0.35	0.35
183.equake	0.97	1.15	1.22
188.ammp	1.02	1.12	1.18
189.lucas	0.40	0.40	0.40

Table B.22: The IPC of an 16-wide 4-cluster processor with consumer requested forwarding. These simulations used the dual-wakeup policy to avoid deadlocks and assumed a 1-cycle detect-to-set latency. These configurations simulated Alpha 21264-like branch prediction and memory dependence prediction.

Benchmark	Detect-to-set delay			
	1	2	3	4
164.gzip	2.37	2.17	1.86	1.73
175.vpr	1.33	1.17	0.80	0.77
176.gcc	1.68	1.63	1.54	1.51
181.mcf	0.11	0.11	0.11	0.11
197.parser	1.79	1.49	1.21	1.15
252.eon	3.10	2.51	1.96	1.86
253.perlbmk	1.15	1.10	1.04	1.01
254.gap	1.66	1.00	0.85	0.79
256.bzip2	3.60	3.34	1.86	1.74
171.swim	0.60	0.60	0.60	0.60
172.mgrid	1.27	1.25	1.17	1.02
173.applu	0.62	0.60	0.58	0.57
177.mesa	4.20	3.16	2.21	2.04
178.galgel	2.83	1.86	1.38	1.30
179.art	0.38	0.38	0.37	0.36
183.equake	3.66	3.18	2.75	2.23
188.ammp	1.68	1.59	1.31	1.25
189.lucas	0.40	0.40	0.40	0.40

Table B.23: The IPC of an 16-wide 4-cluster processor with consumer requested forwarding for different detect-to-set latencies. These simulations used the pro-active operand fetch policy to avoid deadlocks and used mod3 steering. These configurations simulated perfect branch prediction and perfect memory disambiguation.

Benchmark	Detect-to-set delay			
	1	2	3	4
164.gzip	1.80	1.80	1.80	1.80
175.vpr	1.33	1.21	0.99	0.96
176.gcc	1.36	1.36	1.36	1.36
181.mcf	0.11	0.11	0.11	0.11
197.parser	1.59	1.57	1.55	1.52
252.eon	2.50	2.50	2.50	2.50
253.perlbmk	0.99	0.99	0.99	0.99
254.gap	1.33	1.31	1.30	1.29
256.bzip2	3.60	3.57	3.49	3.46
171.swim	0.60	0.60	0.60	0.60
172.mgrid	1.19	1.19	1.18	1.18
173.applu	0.57	0.57	0.57	0.57
177.mesa	2.29	2.29	2.29	2.29
178.galgel	3.41	3.21	3.01	2.86
179.art	0.37	0.37	0.37	0.37
183.equake	3.62	3.60	3.54	3.49
188.ammp	1.50	1.50	1.50	1.49
189.lucas	0.40	0.40	0.40	0.40

Table B.24: The IPC of an 16-wide 4-cluster processor with consumer requested forwarding for different detect-to-set latencies. These simulations used the pro-active operand fetch policy to avoid deadlocks and used load-slice steering. These configurations simulated perfect branch prediction and perfect memory disambiguation.

Benchmark	Detect-to-set delay			
	1	2	3	4
164.gzip	2.34	2.34	2.33	2.32
175.vpr	1.18	1.15	1.13	1.13
176.gcc	1.45	1.45	1.45	1.44
181.mcf	0.11	0.11	0.11	0.11
197.parser	1.75	1.73	1.71	1.69
252.eon	2.99	2.99	2.99	2.98
253.perlbmk	1.05	1.05	1.05	1.05
254.gap	1.53	1.50	1.33	1.31
256.bzip2	3.62	3.62	3.59	3.54
171.swim	0.60	0.60	0.60	0.60
172.mgrid	1.21	1.20	1.19	1.19
173.applu	0.60	0.60	0.60	0.60
177.mesa	3.09	3.09	3.07	3.06
178.galgel	3.52	3.14	2.54	2.37
179.art	0.38	0.38	0.38	0.38
183.equake	2.74	2.74	2.74	2.74
188.ammp	1.60	1.60	1.62	1.58
189.lucas	0.40	0.40	0.40	0.40

Table B.25: The IPC of an 16-wide 4-cluster processor with consumer requested forwarding for different detect-to-set latencies. These simulations used the pro-active operand fetch policy to avoid deadlocks and used dependence steering. These configurations simulated perfect branch prediction and perfect memory disambiguation.

Benchmark	Mod3	Load-slice	Dependence
164.gzip	0.83	0.96	1.02
175.vpr	0.52	0.61	0.62
176.gcc	0.95	0.98	1.00
181.mcf	0.10	0.10	0.10
197.parser	0.79	0.92	0.98
252.eon	0.66	0.70	0.76
253.perlbmk	0.47	0.49	0.53
254.gap	0.58	0.66	0.69
256.bzip2	1.23	1.42	1.52
171.swim	0.60	0.60	0.60
172.mgrid	1.18	1.21	1.19
173.applu	0.55	0.56	0.56
177.mesa	1.05	1.18	1.28
178.galgel	2.62	3.34	3.25
179.art	0.35	0.35	0.35
183.equake	0.98	1.15	1.22
188.ammp	1.03	1.11	1.18
189.lucas	0.40	0.40	0.40

Table B.26: The IPC of an 16-wide 4-cluster processor with consumer requested forwarding. These simulations used the pro-active operand fetch policy to avoid deadlocks and assumed a 1-cycle detect-to-set latency. These configurations simulated Alpha 21264-like branch prediction and memory dependence prediction.

B.4 Hot-register Based Forwarding

Benchmark	Mod3	Load-slice	Dependence
164.gzip	2.03	1.77	2.19
175.vpr	1.14	1.30	1.17
176.gcc	1.64	1.35	1.44
181.mcf	0.11	0.11	0.11
197.parser	1.61	1.55	1.72
252.eon	2.75	2.50	2.95
253.perlbmk	1.07	0.99	1.04
254.gap	1.44	1.32	1.46
256.bzip2	3.51	3.58	3.60
171.swim	0.60	0.60	0.60
172.mgrid	1.26	1.18	1.20
173.applu	0.60	0.57	0.59
177.mesa	3.35	2.27	2.93
178.galgel	2.53	3.17	3.33
179.art	0.38	0.37	0.38
183.equake	3.41	3.39	2.58
188.ammp	1.59	1.51	1.60
189.lucas	0.40	0.40	0.40

Table B.27: The IPC of an 16-wide 4-cluster processor with hot-register based forwarding. These simulations used the pro-active operand fetch policy to avoid deadlocks. These configurations simulated perfect branch prediction and perfect memory disambiguation.

Benchmark	Mod3	Load-slice	Dependence
164.gzip	0.77	0.93	1.00
175.vpr	0.50	0.60	0.61
176.gcc	0.92	0.96	0.98
181.mcf	0.10	0.10	0.10
197.parser	0.78	0.91	0.97
252.eon	0.64	0.69	0.75
253.perlbmk	0.45	0.48	0.52
254.gap	0.55	0.64	0.67
256.bzip2	1.14	1.38	1.45
171.swim	0.60	0.60	0.60
172.mgrid	1.15	1.20	1.17
173.applu	0.54	0.55	0.56
177.mesa	1.00	1.13	1.24
178.galgel	2.40	3.24	2.92
179.art	0.35	0.35	0.35
183.equake	0.94	1.10	1.20
188.ammp	0.98	1.10	1.16
189.lucas	0.40	0.40	0.40

Table B.28: The IPC of an 16-wide 4-cluster processor with hot-register based forwarding. These simulations used the pro-active operand fetch policy to avoid deadlocks. These configurations simulated Alpha 21264-like branch prediction and memory dependence prediction.

B.5 Memory Instruction Steering

Benchmark	Mod3	Load-slice	Dependence
164.gzip	2.59	2.44	2.89
175.vpr	1.51	1.56	1.64
176.gcc	1.69	1.63	1.76
181.mcf	0.11	0.11	0.11
197.parser	1.89	2.01	2.15
252.eon	3.53	3.65	4.00
253.perlbmk	1.39	1.37	1.45
254.gap	2.25	1.92	2.37
256.bzip2	3.73	3.74	3.74
171.swim	0.60	0.60	0.60
172.mgrid	1.27	1.21	1.27
173.applu	0.62	0.59	0.62
177.mesa	4.57	4.15	4.74
178.galgel	2.84	3.41	3.41
179.art	0.38	0.38	0.38
183.equake	3.90	3.92	3.95
188.ammp	1.75	1.82	1.94
189.lucas	0.40	0.40	0.40

Table B.29: The IPC of an 16-wide 4-cluster processor with *ideal* memory-steering. These configurations simulated perfect branch prediction and perfect memory disambiguation. Also, the configurations in these experiments used the CRF method to remove transfer instructions.

Benchmark	Mod3	Load-slice	Dependence
164.gzip	2.37	2.51	2.45
175.vpr	1.34	1.39	1.44
176.gcc	1.68	1.59	1.73
181.mcf	0.11	0.11	0.11
197.parser	1.80	1.79	2.05
252.eon	3.34	3.45	3.83
253.perlbmk	1.28	1.26	1.32
254.gap	1.66	1.38	1.74
256.bzip2	3.72	3.71	3.73
171.swim	0.60	0.60	0.60
172.mgrid	1.27	1.21	1.27
173.applu	0.62	0.59	0.62
177.mesa	4.49	4.10	4.73
178.galgel	2.83	3.66	3.52
179.art	0.38	0.37	0.38
183.equake	3.87	3.91	3.93
188.ammp	1.73	1.73	1.90
189.lucas	0.40	0.40	0.40

Table B.30: The IPC of an 16-wide 4-cluster processor with memory-steering using the last-cluster prediction method. These configurations simulated perfect branch prediction and perfect memory disambiguation. Also, the configurations in these experiments used the CRF method to remove transfer instructions.

Benchmark	Mod3	Load-slice	Dependence
164.gzip	0.89	0.98	1.07
175.vpr	0.55	0.58	0.60
176.gcc	0.96	0.98	0.98
181.mcf	0.11	0.11	0.11
197.parser	0.79	0.88	0.96
252.eon	0.75	0.77	0.83
253.perlbmk	0.54	0.55	0.58
254.gap	0.58	0.64	0.67
256.bzip2	1.58	1.82	1.92
171.swim	0.60	0.60	0.60
172.mgrid	1.18	1.20	1.20
173.applu	0.55	0.56	0.56
177.mesa	1.26	1.41	1.55
178.galgel	2.62	3.24	3.22
179.art	0.34	0.35	0.35
183.equake	1.28	1.44	1.64
188.ammp	1.11	1.20	1.27
189.lucas	0.40	0.40	0.40

Table B.31: The IPC of an 16-wide 4-cluster processor with memory-steering using the last-cluster prediction method. These configurations simulated Alpha 21264-like branch prediction and memory dependence prediction. Also, the configurations in these experiments used the CRF method to remove transfer instructions.

B.6 Critical Operand Steering

Benchmark	Critical Operand
164.gzip	1.00
175.vpr	0.58
176.gcc	1.00
181.mcf	0.11
197.parser	0.98
252.eon	0.71
253.perlbmk	0.52
254.gap	0.70
256.bzip2	1.56
171.swim	0.60
172.mgrid	1.19
173.applu	0.55
177.mesa	1.22
178.galgel	2.63
179.art	0.35
183.equake	1.23
188.ammp	1.12
189.lucas	0.40

Table B.32: The IPC of an 16-wide 4-cluster processor with critical-operand steering. These configurations simulated Alpha 21264-like branch prediction and memory dependence prediction. Also, the configurations in these experiments used the CRF method to remove transfer instructions.

B.7 Issue-width Balance Steering

Benchmark	Issue-balance
164.gzip	2.80
175.vpr	1.46
176.gcc	1.67
181.mcf	0.11
197.parser	2.04
252.eon	3.64
253.perlbmk	1.17
254.gap	1.89
256.bzip2	3.63
171.swim	0.60
172.mgrid	1.27
173.applu	0.62
177.mesa	4.51
178.galgel	3.44
179.art	0.38
183.equake	3.91
188.ammp	1.83
189.lucas	0.40

Table B.33: The IPC of an 16-wide 4-cluster processor with issue-balance steering. These configurations simulated perfect branch prediction and perfect memory disambiguation. Also, the configurations in these experiments used the CRF method to remove transfer instructions.

Benchmark	Issue-balance
164.gzip	1.02
175.vpr	0.62
176.gcc	0.99
181.mcf	0.10
197.parser	0.97
252.eon	0.75
253.perlbmk	0.53
254.gap	0.67
256.bzip2	1.41
171.swim	0.60
172.mgrid	1.21
173.applu	0.56
177.mesa	1.23
178.galgel	3.12
179.art	0.35
183.equake	1.18
188.ammp	1.17
189.lucas	0.40

Table B.34: The IPC of an 16-wide 4-cluster processor with issue-balance steering. These configurations simulated Alpha 21264-like branch prediction and memory dependence prediction. Also, the configurations in these experiments used the CRF method to remove transfer instructions.

Bibliography

[1] Vikas Agarwal. *Scalable Primary Cache Memory Architectures.* PhD thesis, The University of Texas at Austin, May 2004.

[2] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate vs. IPC : The end of the road for conventional microprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, June 2000.

[3] Vikas Agarwal, Stephen W. Keckler, and Doug Burger. The effect of technology scaling on microarchitectural structures. Technical Report TR2000-02, Department of Computer Sciences, The University of Texas at Austin, 2000.

[4] Todd M. Austin and Gurindar S. Sohi. Zero-cycle loads: Microarchitecture support for reducing load latency. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 82–92, December 1995.

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

[6] Ravi Bhargava. *Instruction History Management for High-Performance Microprocessors*. PhD thesis, The University of Texas at Austin, August 2003.

[7] Ravi Bhargava and Lizy K. John. Improving dynamic cluster assignment for clustered trace cache processors. In *Proceedings of the 30th international symposium on computer architecture*, pages 264–274, June 2003.

[8] Bryan Black, Brian Mueller, Stephanie Postal, Ryan Rakvic, Noppanunt Utamaphethai, and John Paul Shen. Load execution latency reduction. In *Proceedings of the 12th International Conference on Supercomputing*, pages 29–36, July 1998.

[9] Eric Borch, Eric Tune, Srilatha Manne, and Joel Emer. Loose loops sink chips. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 299–310, 2002 February.

[10] 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 8th International Symposium on High-Performance Computer Architecture*, pages 289–298, February 2002.

[11] 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, Department of Electrical and Computer Engineering, The University of Texas at Austin, May 2004.

[12] Mary D. Brown, Jared Stark, and Yale N. Patt. Select-free instruction scheduling logic. In *Proceedings of the 34rd International Symposium on Microarchitecture*, pages 204–213, December 2001.

[13] J. Adam Butts and Gurindar S. Sohi. Characterizing and predicting value degree of use. In *Proceedings of the 35th annual international symposium on Microarchitecture*, pages 15–26, 2002.

[14] Ramon Canal, Joan Manuel Parcerisa, and Antonio González. Dynamic cluster assignment mechanisms. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, pages 132–142, January 2000.

[15] Anantha Chandrakasan, William J. Bowhill, and Frank Fox ,editors. *Design of High-Performance Microprocessor Circuits*. IEEE Press, Piscataway, NJ, 2001.

[16] Robert S. Chappell, Francis Tseng, Adi Yoaz, and Yale N. Patt. Difficult-path branch prediction using subordinate microthreads. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 307–317, May 2002.

[17] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the 5th international conference on Architectural support for programming languages and operating systems*, pages 51–61, October 1992.

[18] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[19] José-Lorenzo Cruz, Antonio González, Mateo Valero, and Nigel P. Topham. Multiple-banked register file architectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 316–325, June 2000.

[20] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 266–277, July 2001.

[21] Pradeep K. Dubey and Michael J. Flynn. Optimal pipelining. *J. Parallel Distrib. Comput.*, 8(1):10–19, 1990.

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

[23] Brian Fields, Rastislav Bodik, 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.

[24] Brian Fields, Shai Rubin, and Rastislav Bodik. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 74–85, June 2001.

[25] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-machine multicomputer. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 146–156, December 1995.

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

[27] Manoj Franklin and Gurindar S. Sohi. Arb: A hardware mechanism for dynamic reordering of memory references. *IEEE Trans. Comput.*, 45(5):552–571, 1996.

[28] Daniel H. Friendly, Sanjay J. Patel, and Yale N. Patt. Alternative fetch and issue policies for the trace cache fetch mechanism. In *Proceedings of the 30th annual international symposium on Microarchitecture*, pages 24–33, December 1997.

[29] Heather Hanson, M. S. Hrishikesh, Vikas Agarwal, Stephen W. Keckler, and Doug Burger. Static energy reduction techniques for microprocessor caches. In *Proceedings of the International Conference on Computer Design*, pages 276–283, September 2001.

[30] A. Hartstein and Thomas R. Puzak. The optimum pipeline depth for a microprocessor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 7–13, May 2002.

[31] A. Hartstein and Thomas R. Puzak. Optimum power/performance pipeline depth. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 117–126, December 2003.

[32] Seongmoo Heo, Ronny Krashinsky, and Krste Asanović. Activity-sensitive flip-flop and latch selection for reduced energy. In *Conference on Advanced Research in VLSI*, pages 59–74, March 2001.

[33] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, February 2001.

[34] Ron Ho, Kenneth W. Mai, and Mark A. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, April 2001.

[35] 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, May 2002.

[36] Quinn Jacobson, Steve Bennett, Nikhil Sharma, and James E. Smith. Control flow speculation in multiscalar processors. In *Proceedings of*

the 3rd Symposium on High-Performance Computer Architecture, pages 218–229, February 1997.

[37] Daniel A. Jiménez, Heather L. Hanson, and Calvin Lin. Boolean formula-based branch prediction for future technologies. In *Proceedings of the International Conference on Parallel Architectures and Compilation Technologies*, pages 97–106, September 2001.

[38] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache prefetch buffers. In *Proceedings of the 24th international symposium on computer architecture*, pages 364–373, June 1990.

[39] Norman P. Jouppi and Steven J. E. Wilton. An enhanced access and cycle time model for on-chip caches. Technical Report 93.5, Compaq Computer Corporation, July 1994.

[40] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Cache-line decay: Exploiting generational behavior to reduce leakage power. In *The 28th Annual International Symposium on Computer Architecture*, pages 240–251, July 2001.

[41] R. E. Kessler, E. J. McLellan, and D.A. Webb. The Alpha 21264 microprocessor architecture. In *Proceedings of the International Conference on Computer Design*, pages 90–105, October 1998.

[42] Steven R. Kunkel and James E. Smith. Optimal pipelining in supercomputers. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 404–411, June 1986.

[43] Nasser A. Kurd, Javed S. Barkatullah, Rommel O. Dizon, Thomas D. Fletcher, and Paul D. Madland. Multi-GHz clocking scheme for Intel Pentium 4 microprocessor. In *Proceedings of the International Solid-state Circuits Conference*, pages 404–405, February 2001.

[44] Alvin R. Lebeck, Jinson Koppanalil, Tong Li, Jaidev Patwardhan, and Eric Rotenberg. 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.

[45] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John C. Ruttenberg. The multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, 1993.

[46] Scott McFarling. Combining branch predictors. Technical Report TN-36, DEC Western Research Lab, June 1993.

[47] Ramadass Nagarajan, Karthikeyan Sankaralingam, Doug Burger, and Stephen W. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 40–51, December 2001.

[48] Mario Daniel Nemirovsky, Forrest Brewer, and Roger C. Wood. DISC: Dynamic instruction stream computer. In *Proceedings of the 24th International Symposium on Microarchitecture*, pages 163–171, November 1991.

[49] Koji Nii, Hiroshi Makino, Yoshiki Tujihashi, Chikayoshi Morishima, Yasuhi Hayakawa, Hiroyuki Nunogami, Takahiko Arakawa, and Hisanori Hamano. A low power SRAM using auto-backgate-controlled MT-CMOS. In *International Symposium on Low Power Electronics and Design*, pages 293–298, 1998.

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

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

[52] Sanjay J. Patel, Daniel H. Friendly, , and Yale N. Patt. Critical issues regarding the trace cache fetch mechanism. Technical Report CSE-TR-335-97, The University of Michigan, May 1997.

[53] Matthew A. Postiff, David A. Greene, Gary S. Tyson, and Trevor N.

Mudge. The limits of instruction level parallelism in SPEC95 applications. *Computer Architecture News*, 217(1):31–34, 1999.

[54] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T.N. Vijaykumar. Gated-V_{dd}: A circuit technique to reduce leakage in deep-submicron cache memories. In *International Symposium on Low Power Electronics and Design*, pages 90–95, 2000.

[55] Steven E. Raasch, Nathan L. Binkert, and Steven K. Reinhardt. A scalable instruction queue design using dependence chains. In *Proceedings of the 29th annual international symposium on Computer architecture*, pages 318–329, May 2002.

[56] 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.

[57] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th annual international symposium on Microarchitecture*, pages 24–35, December 1996.

[58] S. Subramanya Sastry, Subbarao Palacharla, and J.E. Smith. Exploiting idle floating-point resources for integer execution. In *Conference on Programming Language Design and Implementation*, pages 118–129, June 1998.

[59] The international technology roadmap for semiconductors. Semiconductor Industry Association, 2001.

[60] André Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 295–306, May 2002.

[61] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, September 2001.

[62] Burton J. Smith. Architecture and applications of the hep multiprocessor computer system. In *Proceedings, SPIE Real Time Signal Processing Architecture*, pages 241–248, August 1981.

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

[64] Eric Sprangle and Doug Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 25–34, May 2002.

[65] Viji Srinivasan, David Brooks, Michael Gschwind, Pradip Bose, Victor Zyuban, Philip N. Strenski, and Philip G. Emma. Optimizing pipelines for power and performance. In *Proceedings of the 35th annual international symposium on Microarchitecture*, pages 333–344, November 2002.

[66] Jared Stark, Mary D. Brown, and Yale N. Patt. On pipelining dynamic instruction scheduling logic. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 57–66, December 2000.

[67] Vladimir Stojanović and Vojin G. Oklobdžija. Comparative analysis of master-slave latches and flip-flops for high-performance and low-power systems. *IEEE Journal of Solid-state Circuits*, 34(4):536–548, April 1999.

[68] John A. Swensen and Yale N. Patt. Hierarchical registers for scientific computers. In *Proceedings of the 2nd International Conference on Supercomputing*, pages 346–354, November 1988.

[69] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Walter Lee, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Saman Amarasinghe, and Anant Agarwal. A 16-issue multiple-program-counter microprocessor with point-to-point scalar operand network. In *Proceedings of the International Solid-state Circuits Conference*, February 2003.

[70] Dean Tullsen, Susan Eggers, and Henry Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 392–403, June 1995.

[71] 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*, pages 81–92, November 1998.

[72] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarsinghe, and Anant Agarwal. Baring it all to software: RAW machines. *IEEE Computer*, 30(9):86–93, September 1997.

[73] Zhenlin Wang, Doug Burger, Kathryn S. McKinley, Steven K. Reinhardt, and Charles C. Weems. Guided region prefetching: a cooperative hardware/software approach. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 388–398, June 2003.

[74] Se-Hyun Yang, Michael D. Powell, Babak Falsafi, Kaushik Roy, and T.N. Vijaykumar. An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance caches. In *International Symposium on High-Performance Computer Architecture*, pages 147–157, 2001.

[75] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive branch prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, November 1991.

[76] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation techniques for improving load related instruction scheduling. In *Proceedings of the*

26th Annual International Symposium on Computer Architecture, pages 42–53, May 1999.

[77] Huiyang Zhou, Mark C. Toburen, Eric Rotenberg, and Thomas M. Conte. Adaptive mode-control: A static-power-efficient cache design. In *International Conference on Parallel Architectures and Compilation Techniques*, 2001.

Index

Vita

Hrishikesh Sathyavasu Murukkathampoondi was born in Coimbatore, India on 15 February 1976 to Sathyavasu Arunachalam and Seshambal Sundareswaran. He received a Bachelor of Engineering degree in Electronics and Communication Engineering from the University of Madras in May 1997. The following fall he entered the graduate program in Computer Engineering at the University of Texas at Austin. In August 1999 he received a Master of Science degree and subsequently joined the Ph.D. program in Computer Engineering at the University of Texas at Austin. Part of his graduate research was supported by an Intel Foundation Ph.D. fellowship (2002-03).

Permanent address: Flat No. 4, 87 Greenways Road
Chennai, India 600028

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.