

University of California  
Santa Barbara

Performance Tradeoffs in  
Multistreamed Superscalar Architectures

A Dissertation submitted in partial satisfaction  
of the requirements for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

by

Mauricio Jose Serrano

Committee in charge:

Professor Roger C. Wood, Chairman  
Professor Michael Melliar-Smith  
Professor Louise Moser  
Professor Steven Butner  
Dr. Mario Nemirovsky

March 1994

The dissertation of Mauricio Jose Serrano is approved

---

---

---

---

---

Committee Chairperson

March 1994

To my family and my friends

## **Acknowledgments**

The author wishes to express his gratitude to Dr. Wood for his supervision and support of this work. He also wishes to thank Dr. Parhami and Ching Yu-Hung for supplying the computer resources needed for his work. Special thanks to our work-team in computer architecture. Section 4.1.3 was developed with Wayne Yamamoto and Mario Nemirovsky.

The author is also grateful to Mario Nemirovsky, Louise Moser, Michael Melliar-Smith, and Steve Butner for serving on his Ph.D. committee.

This research has been supported by the State of California MICRO grants #90-185, Delco Systems Operation of General Motors Corp., #92-178 and #93-169 from Apple Computer Inc., and by an unrestricted gift from Apple Computer Inc.

Mauricio J. Serrano  
University of California Santa Barbara  
March, 1994.

## VITA

Mauricio J. Serrano was born in Cali, Colombia, on September 10, 1958. He received the Electrical Engineering degree in electrical engineering from the Universidad Javeriana, Bogota, Colombia, in 1982. He worked from 1982-1984 in software development. In 1984 he obtained a Fulbright Fellowship to pursue graduate studies in the U.S. He obtained the M.S. degree in computer engineering from Rensselaer Polytechnic Institute, Troy, NY, in 1986. He returned to Colombia where he taught from 1986-1990 at Universidad Javeriana, Bogota and Cali. He returned to the U.S. in 1990 to pursue the Ph.D. degree in computer engineering from The University of California Santa Barbara. His current research interests include parallel and distributed processing, computer architecture, performance modeling, and simulation.

## PRINCIPAL PUBLICATIONS

- M.J. Serrano, W. Yamamoto, R. Wood, M. Nemirovsky, "A Model for Performance Estimation in a Multistreamed Superscalar Processor," *Seven International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, May 1994, Vienna, Austria, Lecture Notes in Computer Science, Springer-Verlag.
- M.J. Serrano, B. Parhami, "Optimal Architectures and Algorithms for Mesh-Connected Parallel Computers with Separable Global Buses," *IEEE Transactions of Parallel and Distributed Systems*, October 1993, pp. 1073-1081.
- M.J. Serrano, D. Donalson, R. Wood, M. Nemirovsky, "DISC: Dynamic Instruction Stream Computer, an evaluation of performance," *Proceedings of the Hawaii Conference on Systems and Science*, Hawaii, January 1993, pp. 183-192.

## ABSTRACT

Performance Tradeoffs in Multistreamed Superscalar Architectures

by

Mauricio Jose Serrano

Superscalar processors employ multiple functional unit designs that can dispatch several instructions every cycle. Two factors limiting the number of instructions dispatched per cycle are: 1) the number of functional units available (hardware), and 2) the amount of parallelism in the workload (software). While the number of functional units determines the peak throughput of a processor, the instruction-level parallelism determines the actual performance obtained. Data dependencies and control breaks constrain instruction-level parallelism resulting in a sustained system performance that is well below the peak.

The ability to execute simultaneously multiple instruction streams, referred to as multistreaming, significantly increases the number of independent instructions that can be issued in a cycle. A multistreamed, superscalar processor can dispatch instructions from multiple streams simultaneously. Each stream context is stored internally. The processor adjust the scheduling policy as the workload changes to maximize throughput.

This dissertation explores a methodology for the design of a distributed multistreamed-superscalar processor that addresses instruction issue, implementation of precise interrupts, speculative execution, scheduling, and cache sharing. One of the goals of the design of a processor architecture is to minimize the number of global interconnections and reduce the need for clock synchronization. Variants in the design are possible because of tradeoffs that can be done in the design. We explore several problems present in multistreamed architectures and discuss possible solutions.

We present an analytic model to estimate the overall performance of multistreamed architectures. The model uses simple workload and architectural descriptions that are obtained using commonly available tools. The model produces instructions executed per cycle (IPC) as the number of streams is varied.

## TABLE OF CONTENTS

1	Introduction .....	1
2	Dynamic Superscalar Multistreamed Architectures .....	6
	2.1 Background.....	6
	2.1.1 Multithreaded Architectures.....	6
	2.1.2 Examples of Recent Multithreaded Processors .....	11
	2.1.3 Superscalar Machines .....	13
	2.2 Dynamic Multistreamed Superscalar Processors .....	15
	2.2.1 Decoupling Control and Data .....	17
	2.2.2 Data Type Locality Principle .....	18
	2.2.3 Scheduling Instructions .....	19
3	Performance Evaluation of Multistreamed Architectures .....	22
	3.1 A Multistreamed Simulator for the Power Architecture .....	22
	3.1.1 The Power Architecture.....	22
	3.1.2 Powersim .....	23
	3.1.3 A High Level Model of Instruction Execution.....	24
	3.1.4 Hardware Simulation in Powersim.....	25
	3.2 Preliminary Study .....	27
	3.2.1 Effect of Configuration .....	28
	3.2.2 Effect of Memory Latency .....	32
	3.2.3 Effect of Schedule .....	34
4	Performance Estimation of Multistreamed Architectures .....	37
	4.1 The Model .....	37
	4.1.1 Architecture Characterization.....	38
	4.1.2 Workload Characterization.....	39
	4.1.3 Model of Structural Hazards .....	39
	4.1.4 Model of Data and Control Hazards .....	42
	4.1.5 Rationale .....	43
	4.2 Saturation Point .....	44
	4.2.1 Extended Workload Specification Parameters .....	44
	4.2.2 Homogeneous Workload.....	45
	4.2.3 Best-Case Schedule .....	46
	4.2.4 Fair Schedule .....	47
	4.3 Validation .....	50
	4.3.1 Benchmarks .....	51
	4.3.2 Comparison to Simulation Results .....	52
	4.3.3 Discussion .....	52

5	A Multistreamed Instruction Issue Mechanism .....	58
5.1	Background .....	59
5.2	Components of the Issue Mechanism .....	62
5.2.1	Pipeline Structure .....	62
5.2.2	Instruction Tag Unit .....	62
5.2.3	General Block Diagram .....	63
5.2.4	The Result Buffer .....	65
5.2.5	Functional Units .....	67
5.2.6	Stream Window .....	68
5.2.7	Determining the Register Conflict Set .....	69
5.3	Issue Mechanism .....	70
5.3.1	Issue Rules .....	70
5.3.2	Instruction Issue Format .....	72
5.3.3	Register Renaming .....	72
5.3.4	Control Hazards .....	73
5.3.5	Memory Hazards .....	73
5.3.6	Exception Rules .....	75
5.4	Relaxing Precise Interrupts .....	76
6	Performance Tradeoffs of Multistreamed Architectures .....	78
6.1	Scheduling Instructions .....	78
6.1.1	Instruction Network .....	79
6.1.2	Single Stream Arbitration .....	80
6.1.3	Multistream Arbitration .....	82
6.1.4	Reducing the Critical Path .....	83
6.1.5	A Look-Ahead Scheme for Arbitration .....	83
6.2	Multithreaded Cache .....	86
6.2.1	Understanding Cache Degradation .....	88
6.2.2	Cache Behavior for a Small Number of Contexts .....	90
6.2.3	Victim Cache .....	94
7	Conclusion .....	99
7.1	General Architecture .....	99
7.2	Performance Evaluation .....	100
7.3	Performance Estimation .....	100
7.4	Multistreamed Instruction Issue Mechanism .....	101
7.5	Performance Tradeoffs .....	102
	BIBLIOGRAPHY .....	103

## TABLE OF ILLUSTRATIONS

Table 1.1. Area percentages of the i860 chip. ....	5
Table 3.1. Principal commands of Powersim .....	23
Table 3.2. Powersim's hardware specification parameters. ....	27
Table 3.3. SPEC '89 instruction mixes. ....	29
Table 3.4. Configurations evaluated. ....	29
Table 3.5. Multistream improvement with a random schedule. ....	32
Table 3.6. Multistreaming improvement for memory latency 4, random schedule and configurations C1, C2, C3. ....	34
Table 4.1. Architectural specification parameters. ....	39
Table 4.2. Workload characterization parameters. ....	39
Table 4.3. State table for the example in Figure 4.1 .....	41
Table 4.4. Extended workload characterization parameters. ....	45
Table 4.5. Workload characteristics. ....	51
Table 4.6. Results for homogeneous workloads. ....	53
Table 4.7. Results for heterogeneous workloads. ....	54
Table 4.8. Error for homogeneous workloads. ....	54
Table 4.9. Error for heterogeneous workloads. ....	55
Table 4.10. SPEC'89 characteristics. ....	57
Table 5.1. Classification of instructions by interrupts. ....	76
Table 5.2. Example of tagging blocks of instructions. ....	77
Table 6.1. Average number of entries in the victim cache to produce the same effect as increasing the cache associativity for one, two and four streams. ....	98

## LIST OF FIGURES

Figure 2.1. Examples of blocked, static, and dynamic multithreading. ....	9
Figure 2.2. Utilization of the multithreaded processor. ....	9
Figure 2.3. Block diagram of a multistreamed, superscalar processor. ....	15
Figure 2.4. Detailed hardware organization. ....	16
Figure 2.5. Decoupling control and data. ....	17
Figure 2.6. Two approaches for the data network .....	18
Figure 2.7. Example of dynamic scheduling instructions from different threads using round-robin. ....	21
Figure 3.1. Functional blocks of the Power architecture. ....	23
Figure 3.2. Functional block diagram of Powersim. ....	24
Figure 3.3. Operation of the Dispatch Stack. ....	26

Figure 3.4. Statistics for configuration C1 (top), C2 (middle), and C3 (bottom), with random (fair) schedule. ....	31
Figure 3.5. Statistics for configuration C1 (top), C2 (middle), and C3 (bottom) with random (fair) schedule and memory latency 4. ....	33
Figure 3.6. Performance of each stream for configuration C3, prioritized schedule, and memory latency of 1. ....	35
Figure 3.7. Performance of each stream for configuration C3, prioritized schedule, and memory latency of 4. ....	35
Figure 4.1. Markov chain and the state transition matrix (STM) for the example with homogeneous workloads, $w=3$ , $M3=[0,3]$ , $C=(1,2)$ . ....	41
Figure 4.2. Example of optimization with two benchmark types. ....	47
Figure 4.3. Scaled instruction mix for two benchmarks. ....	49
Figure 4.4. Comparing simulation and prediction with homogeneous workloads for the SPEC benchmarks. ....	56
Figure 5.1. Hardware implementation for Tomasulo's algorithm. ....	59
Figure 5.2. Example of tags in the instruction window. ....	63
Figure 5.3. Example of multistreamed hardware. ....	64
Figure 5.4. The multistreamed result buffer. ....	65
Figure 5.5. Block diagram of functional units. ....	67
Figure 5.6. The stream window ....	68
Figure 5.7. Computing the WAW and RAW register conflict set for a small window. ....	70
Figure 5.8. Example of dependency graph for a window of four instructions. ....	71
Figure 6.1. The scheduling network. ....	80
Figure 6.2. Wavefront computation cell for port arbitration. ....	81
Figure 6.3. Port arbitration in a multistreamed processor. ....	83
Figure 6.4. Wavefront computation of the C vector. ....	85
Figure 6.5. Simulation of the cache with a model. ....	89
Figure 6.6. Relative improvement over one thread for two (a,b) and four threads (c,d), for caches with associativity one (a,c), and four (b,d). ....	91
Figure 6.7. Average percentage of bus utilization for multithreaded caches with associativities 1, 2, and 4, and one, two, and four threads. ....	93
Figure 6.8. Average interference for two and four streams and associativities 1, 2, and 4. ....	94
Figure 6.9. Organization of victim cache. ....	95
Figure 6.10. Percentage of improvement in performance for a 32K direct mapped cache as a function of the number of entries in a victim cache. ....	97
Figure 6.11. Average percentage of improvement as a function of the cache size (in kilobytes), for 2, 4, 8, and 15 entries in the victim cache. ....	97

# 1 Introduction

As existing computing systems increase in computational power, there is an increasing demand for still higher computational power. Computer architects are continuously challenged by the desire for higher computational power, accommodating the existing implementation technology.

Advances in VLSI and compilers supported the development of RISC (Reduced Instruction Set Computers) which execute most instructions in one cycle. The primary advantage comes from a reduction of control logic. Simple control reduces the clock cycle time because the processor can be simplified and easily pipelined. RISC designs employ less chip area in control logic and more in bigger register files to reduce the memory traffic.

Approaches to obtaining higher performance from applications include executing several instructions simultaneously or using several processors to execute a task in a multiprocessor system. We classify parallelism in a multiprocessor system into coarse-grain (task-level), medium-grain (thread-level), and fine-grain (instruction-level).

Multiprocessor systems employ task-level parallelism. Tasks execute concurrently in several processors. Tasks communicate using communication primitives provided by the hardware. At present, the most popular scheme used in multiprocessors is the shared memory paradigm. The reason is that parallel algorithms are expressed in a clean way when the algorithm assumes that the data can be accessed by all the processors. Memory is usually distributed among the processors, with each processor owning a piece of the global shared memory space, as in the DASH prototype [leno92]. This means that while local accesses take few cycles, many remote accesses may take hundreds of cycles.

Present day multiprocessor systems do not achieve high processor utilization because of network contentions, memory latency, and synchronization overhead. These problems limit the number of applications that run efficiently in a multiprocessor. If there is a way to increase the utilization of processors, simpler multiprocessor systems achieving the same level of performance could be implemented.

Asynchronous dataflow machines employ fine-grain parallelism. Programs execute in a data-driven manner to exploit all the potential parallelism. However, a parallel program

performs more work than a sequential one for the same problem because of the generation and synchronization of parallel activities. There is a high overhead in dataflow execution, directly attributable to asynchronous parallel execution [arvi88].

Thread-level parallelism is a compromise solution between dataflow execution and traditional von Neumann architectures. Several authors [cull92,nikh92] proposed compilation models to allow additional instruction sequencing constraints, making explicit the notion of thread. Unlike processes, threads are short lived; their life is between a few instructions to hundreds of instructions. Thus, the parallelism granularity of threads is coarser than in dataflow machines; there is no need for synchronization at the instruction level. Since the synchronization constraints are far fewer than in dataflow machines, conventional processors can exploit threads [cull92]. Thread parallelism is finer than task-parallelism. For example, many threads could exist in the same protection domain; therefore, thread switching does not need a costly context switching. The Threaded-Abstract-Model (TAM) [cull91] is a recent work in thread parallelism.

Conventional processors (von Neumann) use pipelining of resources to achieve high performance. However, hazards caused by data and control dependencies result in pipeline stalls or idle periods when new instructions cannot be issued. If a source register of an instruction is the same as the target register of a previous instruction, a true data dependency exists (RAW: Read-after-Write). Other data dependencies include anti-dependencies (WAR: Write-after-Read) and output-dependencies (WAW: Write-after-Write) [henn90].

Whenever a data dependency occurs, the dependent instruction cannot issue until the resolving instruction finishes execution. A conditional branch causes a control dependency if the branch depends on an operand produced by an instruction executing in the pipeline. Instructions following the branch stall until the condition is resolved. High latency in memory operations also causes idle cycles in the processor. Studies in shared memory systems show processor efficiency as low as 30% for some applications [webe89].

Today, most processors use multiple issue to achieve high performance. In a superscalar processor, dynamic instruction scheduling is an essential part of the processor. If out-of-order issue is allowed, dependent instructions do not block the flow of all subsequent instructions. The dynamic instruction scheduler checks data dependencies at run-time and issues instructions that are free of data dependencies regardless of program order.

Steady advances in circuit technology have decreased cycle times but have not improved performance at the rate demanded by advanced applications. At present, most of the

processors designed can issue more than one instruction per cycle. However, there are two factors that limit the number of instructions executed in a cycle. One is the number of functional units available (hardware) and the other is the amount of parallelism in the workload (software). While the number of functional units determines the peak throughput of a processor, the level of instruction-level parallelism limits the actual performance obtained. Data dependencies and control breaks constrain instruction-level parallelism resulting in a sustained system performance that is well below the peak.

Combining hardware and compiler techniques can increase the level of parallelism. Some of the hardware techniques that increase parallelism by reducing hazards are register renaming, out of order execution, branch prediction, and speculative execution. On the compiler side, loop unfolding and instruction reordering are some of the techniques employed [henn90]. However, the true data dependencies determine the upper bound to the performance. Satisfying this upper bound requires many resources resulting in poor resource utilization for most applications. As the number of functional units increases within processors, it is unlikely that a single stream can produce enough parallelism to effectively exploit additional resources.

The instruction-level parallelism of many programs is not evenly distributed over time. Programs have sections that can produce high parallelism exploitable with an adequate configuration. However, the serial sections of a program exhibit low performance with the corresponding poor resource utilization for the periods of low issue.

The ability to execute multiple instruction streams simultaneously, referred to as *multistreaming*, significantly increases the number of independent instructions issued per cycle. A multistreamed computer is a multithreaded computer that sustains several instruction streams simultaneously. A multistreamed, superscalar processor can dispatch instructions from multiple streams simultaneously (i.e., in the same cycle). By storing each stream context internally, the scheduler can select independent instructions from all active streams and dispatch the maximum number of instructions on every cycle. A dynamically interleaved processor adjusts the scheduling policy as the workload changes to maximize throughput. For example, the sequencer schedules instructions from a given stream using a priority scheme and, when the selected stream is not ready, its slot is dynamically reallocated to the ready ones. If the active streams are independent, then the total number of instructions dispatched per cycle will increase as the number of active streams increases.

The number of transistors contained on a single chip has increased with the advance of microelectronics technology. We see single chip processors that contain several million transistors. It is expected that a single chip can integrate several processors. In particular, it seems attractive to have a multithreaded, multistreamed processor integrated into a chip.

As device dimensions scale down, interconnections become a major concern in high-performance integrated circuits because the capacitance and resistance of wires increase rapidly as chip size grows larger and minimum feature size is reduced [bako90]. Gate delays decrease with the reduction in the feature size. Local interconnections do not pose a serious performance problem because of their short lengths. The problem is that the delay of a global interconnection (such as those extending from corner to corner on a die) increases in comparison to a gate delay. Rising wire resistance not only degrades the signal propagation delay but also makes designing a good power distribution network very difficult. For example, in larger submicron CMOS chips, a 1 cm long aluminum interconnection in a submicron technology could have a distributed time delay constant of 10 ns, which is much larger than the sub-nanosecond gate delays. Obviously, measures must be taken to avoid such large distributed RC delays.

Long interconnection delays have implications on the architecture design of a processor. The architecture should minimize the number of global interconnections. Architecture modularity should emphasize keeping most interconnections local to reduce global interconnection. In some cases, it is better to replicate simple hardware modules rather than sharing the module. For example, some designs use the integer unit to compute the effective address for a memory operation as well as for normal ALU operations. To minimize the communication of the two unit types, the load/store unit could contain an additional adder.

Integrating a multistreamed processor on a single chip provides better utilization of the processor hardware by fine grain hardware sharing. Hardware utilization increases by sharing several functional units among several independent instruction streams. The context state storage such as the PC, the register file, and the control registers cannot be shared, but any operational functional blocks such as the instruction decoder, ALU, FPU, and the bus interface can be shared. However, shared resources incurs extra overhead cost for scheduling and contention, so an optimal configuration of the hardware needs a detailed cost-area-performance analysis. For example, Table 1.1 shows area measurements for the i860 processor chip [kowa89]. If we were to build a multistreamed processor from this area estimation, most of the advantage would come from sharing the floating point unit and the

cache, since the overhead of sharing could be less than the cost of duplicating these functional blocks; also cache coherence can be maintained more easily if the cache is shared. Each stream could have at least one private integer unit, since they do not occupy much area. We should also concentrate on minimizing the global wires to maximize clock speed. The interface to the external world (bus) is a good candidate for sharing, since it takes a significant area and development cost.

Function block	Area
Floating point unit	26 %
Cache	26 %
Bus drivers and receivers	22 %
Global wires	9.1 %
Integer Unit	8.6 %
Others	8.3 %

**Table 1.1.** Area percentages of the i860 chip.

Finally, some recommendations. In developing a new processor, the design time and simplicity are regarded as significantly more important than the optimal design under given constraints. As design automation progresses and circuit-level or layout-level libraries for functional blocks are accumulated, more attention will be shifted from the design simplicity with reasonable performance to the most optimal design [park91]. Factors to consider are the silicon area and power consumption of the chip. Since silicon area can be traded for speed and the power consumption of a functional block is usually proportional to the silicon area, the silicon area is a good first-order cost function considered in previous studies [park91].

This dissertation concentrates on the hardware support to obtain high-performance from multithreaded parallelism. Chapter 2 discusses the architectural principles needed to build a high-performance multistreamed architecture. Chapter 3 presents a performance evaluation tool used for a preliminary study of the architecture. Chapter 4 introduces an analytic model using a combination of simulation and analysis to predict performance. Chapter 5 discusses the problems of sharing functional units, cache, scheduler hardware; it presents a solution that contemplates interrelated aspects such as precise interrupts, branch speculation, technology issues, and multiple instruction issue from several streams. Finally, Chapter 6 presents the conclusions and the recommendations for future work ●

## 2 Dynamic Superscalar Multistreamed Architectures

This chapter explains the basic concepts of dynamic superscalar multistreamed architectures. Superscalar processors can issue more than one instruction per cycle. Multistreamed superscalar processors can dispatch instructions from multiple streams in the same cycle.

A Dynamic Superscalar Multistreamed Processor is a multistreamed, superscalar processor that adjusts the scheduling policy as the workload changes to maximize throughput. The processor can run in single or multiple stream mode, according to the number of threads active. When a single stream is running, the processor executes as many instructions as dependencies and the configuration allow. As the number of streams increases, the issue scheme changes to accommodate instructions from several streams at the same time.

### 2.1 Background

#### 2.1.1 Multithreaded Architectures

A multithreaded processor allows several threads to run on a single processor. In a multithreaded architecture, each physical processor supports some number of instruction streams or threads, each of which is programmed essentially like a traditional sequential processor. A thread can be defined as a light-weight process, with a life span ranging from a few to thousands of instructions.

Multithreaded computation models have been proposed for very diverse architectures, from conventional processors with software-supported multithreaded to specialized architectures with total hardware support [cull91]. To understand the differences between the different forms of architectural support for multithreading, we classify them in two major categories: coarse-grain, and fine-grain.

Coarse-grain (blocked) multithreading switches a thread out of the processor if a long-latency memory or synchronization operation is issued [hals88,nikh92]. Another thread is context-switched in while the long-latency operation executes, so the processor does not go

unused for that time. For this scheme to be useful, the latency of context switching must be less than the latency of the operation.

Fine-grain (interleaved) multithreading interleaves cycle-by-cycle instructions from different threads. The processor supports several streams in hardware; a set of running threads may use a hardware stream. Multithreading based on multistreaming reduces the hazards in a processor by simultaneously scheduling instructions from independent streams. This increases the issue rate and the utilization of resources. A multithreaded processor can also reduce the idle time caused by long memory latencies by scheduling alternate threads while the memory operation is being performed, without performing costly context-switching. On the other hand, this scheme relies on maintenance of simultaneous contexts (streams). This leads inevitably to a large number of registers to sustain these contexts. The overhead introduced may reduce the maximum achievable clock rate; however, a careful design can reduce this effect.

An example of fine-grain multithreading is instruction interleaving in a pipeline. A pipeline is interleaved if an instruction from a different instruction stream enters the pipe at every cycle, and there are at least as many instruction streams as pipe stages. Pipeline cycles are not wasted when a stream starts a memory operation because other streams can take advantage of those cycles. Instruction interleaving was introduced in the peripheral processor of the CDC6600 [thor64] to hide high latency memory operations. Ten processes were interleaved in a cyclic way in the pipeline and the instruction cycle time was equal to the memory access time. This scheme also eliminates hazards from pipeline execution by interleaving independent instructions from different streams.

It is important to distinguish the ideas of *thread* and *stream*. *Thread* relates to software, while *instruction stream* (IS) relates to hardware. IS refers to the hardware support required to sustain software threads that share many resources, such as the stream's register file. In other words, for a thread to become active, a IS must accept it. We say that a *thread* is allocated to a processor *stream*. In this context, interleaved multithreading requires support for several ISs, blocked multithreading supports only one IS, and both may support multiple threads in software. In any case, the number of ISs implemented is less than or equal to the number of threads.

Multistreamed machines employ different scheduling schemes. The Denelcor HEP computer [smit81] interleaves several streams using *static* scheduling. The HEP pipeline multiplexes eight streams; thus, each stream has a fixed  $1/8$  slot partition of the processor.

This leads to waste of throughput when a stream cannot make use of its time. *Dynamic* scheduling assigns time to each stream according to a static partition, but the time can be reassigned to active streams if a stream does not make use of its time [nemi90,pras91]. Thus, the scheduler dynamically reallocates time when a thread blocks.

Figure 2.1 illustrates the coarse-grain (blocked) and fine-grain (static and dynamic) multithreading schemes on a single-issue processor; we use two streams in our example. We assume that a single thread runs on a single stream. The example assumes a context switch cost of one cycle, data dependency stalls of one cycle, and a memory latency of three cycles. In blocked multithreading, thread *A* switches out of the processor when it issues a high latency memory instruction, and thread *B* switches in and starts running. Thread *A* resumes when thread *B* issues a memory instruction. Note that a data dependency between the instructions  $R_1 \leftarrow R_2 + R_3$  and  $R_4 \leftarrow R_1 + R_3$  in thread *B* stalls the pipeline for one cycle. In static interleaving, threads *A* and *B* share the processor in a multiplexed manner; Thread's *A* slot is wasted while its memory access is performed. In dynamic interleaving, thread *B* can use thread's *A* slot while *A* is idle.

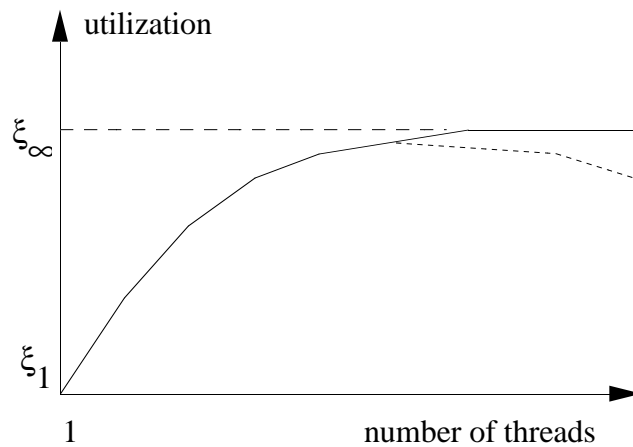
It is important to understand the performance of each of the multithreading forms discussed so far. Let  $R$  be the average number of instructions executed until a memory instruction appears in a thread. Let  $I$  be the average number of pipeline stalls (idle) produced executing these  $R$  instructions; thus,  $R+I$  is the total number of cycles until a memory instruction appears. Let  $L$  be the average memory latency in cycles. Therefore, the processor efficiency in a conventional processor (single thread, single stream) is

$$\xi_1 = \frac{R}{R+I+L}$$

We expect that multithreading will improve the utilization of a processor with the number of threads as shown in Figure 2.2. The utilization of the single-thread, single-stream processor is defined as  $\xi_1$ . The utilization increases with the number of threads, but it saturates at some value  $\xi_\infty$  for a number of threads sufficiently high. In practice, the utilization of a multithreading machine could degrade as the number of threads increases, as suggested by the dotted lines. The degradation analysis will be seen in Chapter 5.

cycle	Coarse-grain (blocked)		Fine-grain (static interleaved)		Fine-grain (dynamic)	
	Thread A	Thread B	Thread A	Thread B	Thread A	Thread B
1	$R_1 \leftarrow R_2 + R_3$		$R_1 \leftarrow R_2 + R_3$		$R_1 \leftarrow R_2 + R_3$	
2	Load $R_2$			$R_1 \leftarrow R_2 + R_3$		$R_1 \leftarrow R_2 + R_3$
3	CONTEXT SWITCH ->		Load $R_2$		Load $R_2$	
4		$R_1 \leftarrow R_2 + R_3$		$R_4 \leftarrow R_1 + R_3$		$R_4 \leftarrow R_1 + R_3$
5		STALL	STALL			$R_7 \leftarrow R_1 + R_8$
6		$R_4 \leftarrow R_1 + R_3$		$R_7 \leftarrow R_1 + R_8$	$R_1 \leftarrow R_4 + R_5$	
7		$R_7 \leftarrow R_1 + R_8$	$R_1 \leftarrow R_4 + R_5$			Load $R_2$
8		Load $R_2$		Load $R_2$		
9	<- CONTEXT SWITCH					
10	$R_1 \leftarrow R_4 + R_5$					

**Figure 2.1.** Examples of blocked, static, and dynamic multithreading.



**Figure 2.2.** Utilization of the multithreaded processor.

In coarse-grain multithreading a thread executes on the average  $R$  instructions until a memory request is found. Assuming a thread is ready, the processor switches to a new thread after a switch delay of  $C$  cycles ( $C < L$ ). If no thread is ready, the processor idles. If *busy*, *switching*, and *idle* are the times the processor spends in the corresponding states, the utilization is [cull92]

$$\xi = \frac{\textit{busy}}{\textit{busy} + \textit{switching} + \textit{idle}}$$

If there is a sufficiently large number  $N$  of threads, then the  $L$  cycles for a memory request will elapse before all the other ready threads have a turn at the processor. Thus, the memory latency is entirely masked and the utilization of the processor is determined by the run length and the switch delay. The processor saturates, since increasing the number of threads has no effect on performance. The utilization in saturation is

$$\xi_{\textit{blocked},\infty} = \frac{R}{R + I + C}$$

In the static interleaved scheme, the pipeline multiplexes  $S$  streams. The utilization is just  $\xi_{\textit{static},1} = 1/S$  (for large  $S$ ) when only one thread is running. The efficiency increases with the number of threads but does not reach unity, because slot times are wasted while memory or synchronization operations execute. Thus, the maximum utilization is

$$\xi_{\textit{static},\infty} = \frac{R}{R + L}$$

In the dynamic interleaved scheme, the utilization for a single thread is the same as in the conventional processor since the throughput is reallocated to the single thread. Thus,  $\xi_{\textit{dynamic},1} = \xi_1$ . Assuming a large number of streams  $S$ , the processor will reallocate throughput if a stream cannot make use of a time slot. Thus, the theoretical limit of utilization using an ideal scheduler approaches one

$$\xi_{\textit{dynamic},\infty} = 1$$

From our discussion, we derive several conclusions

- The primary benefit of blocked multithreading is to hide memory latency, provided that the context switching cost  $C$  is lower than the memory latency  $L$ . A possible application target is modern multiprocessor systems with distributed shared-memory. For example, the worst case of access for a remote memory module in the DASH multiprocessor [leno92] is 132 cycles. Another application is real-time systems where the access time to sensory information is high. A problem with modern conventional processors is the high cost of context switching; however, some attempts have been made with the SPARC [hida93].

- The primary benefit of static interleaving is to hide data and control hazards, if there is a sufficient number of active threads. The performance of the processor running a single thread is horrendous, which excludes this scheme from many applications dominated by Amdahl's law.
- Dynamic interleaving gives the best performance for any given number of threads. However, it implies more complex hardware that comes of holding more contexts and implementing a dynamic scheduler. The additional overhead could also reduce the maximum achievable clock speed.

We can see that the relative order of the hardware costs for the three schemes is:

$$COST_{blocked} < COST_{static} < COST_{dynamic}.$$

This discussion raises the following questions: What is the cost and performance of a multistreamed architecture ? What is the expected performance of a combined workload running on a multistreamed computer ? Is there an optimal number of threads ? We will address these questions in this thesis.

Finally, consider the dynamic scheme. There are other schemes that achieve a processor utilization close to unity. For example, the Tera Computer [smit90] and DART [shet91] can execute multiple instruction streams simultaneously. On every tick of the clock, a stream that is ready to execute is selected and allowed to issue. When an instruction finish, the stream to which it belongs becomes ready to execute the next instruction. The utilization of the processor can reach 100% if there are enough instruction streams in the processor such that the average instruction latency is filled with instructions from other streams. However, the single thread performance is horrendous, as in the static scheme with one thread.

Multithreading is a technique that is orthogonal to superscalar or superpipelining. Thus, multithreading adds a degree of freedom to the design of a processor.

### 2.1.2 Examples of Recent Multithreaded Processors

The MASA architecture proposed by Halstead et al [hals88] is intended as a basic building block for a shared-memory multiprocessor that can execute parallel Lisp programs efficiently. MASA is an interleaved architecture in which each processor can have several active tasks loaded at once and can switch from one to another on every clock cycle. Tolerance to communication improves by allowing other tasks to run while some are blocked, even if only briefly. There are several task context frames, each consisting of a set

of process state registers and a set of general purpose registers for each resident process. One task frame is selected among the *ready* Tasks, and its state is changed to *running*. Instructions are selected from the *running* Tasks. In MASA, like in HEP, an instruction of a process is not issued until the previous instruction of that process is completed. However, MASA employs dynamic scheduling, while HEP uses round-robin scheduling.

MASA uses parallel streams for lightweight procedure calls and traps. This feature is supported by its bank of registers shared in the style of register windows. Children streams have their own registers and share several registers with their parents. When a trap occurs, a trap handler is installed on a new stream. The new stream is allowed to probe the context of the trapped stream. Similarly, by creating a new stream for each procedure invocation, parameters of the procedure are passed in the shared registers automatically.

TERA [alve90,smit90], the successor of HEP, is a shared-memory multiprocessor with multi-stream processors and interleaved memory units interconnected by a packet-switched pipelined interconnection network. Each processor has support for 16 protection domains and 128 streams. A protection domain contains an independent set of registers holding stream resource limits, accounting information, and a memory map. Therefore, each processor can execute 16 tasks in parallel, and can interleave 128 streams. Streams have their own register set and are hardware scheduled. On every tick of the clock, the processor logic selects an instruction from a ready stream. When an instruction completes, the stream to which it belongs becomes ready to execute the next instruction. The processor is fully utilized provided that there are enough instruction streams in the processor such that the average latency is filled with instructions from other streams. TERA employs a VLIW wide instruction format, packed with three operations: Memory, Arithmetic, and Control.

\*T is a blocked multithreaded architecture for massively parallel systems using a pipelined interconnection network [nikh92]. This processor is a descendant of dataflow architectures such as P-RISC and Monsoon [nikh89,cull90], but unlike them, \*T attempts to be compatible with conventional von Neumann processors. The architecture concentrates on an efficient context switch mechanism and on an efficient split-phase communication mechanism for threads with a typical life span of a few tens of instructions. Messages carry continuations, i.e. an identifier for the appropriate thread which is reactivated on the arrival of the response. As in TERA, many threads are needed to avoid idling the processor if the memory latency is long. The processor is separated into three coprocessors: 1) a remote memory request coprocessor in charge of replying to incoming messages, 2) a

synchronization coprocessor in charge of preparing and receiving messages, and 3) the data coprocessor.

DISC is an interleaved pipelined multithreaded processor that addresses the need for faster response to hard-deadline real-time requests, while still providing efficient operation of non-critical threads [nemi90,nemi91,serr93]. Its pipeline can run from a single instruction stream (IS) to multiple instruction streams. The processor throughput can be allocated among ISs, and can be dynamically reallocated when an IS scheduled to run is not ready. Scheduling of streams on an instruction basis using a hardware scheduler allows simple partitioning of the processing power among the several active real time tasks. It is thus possible to assign an interrupt to a given stream which begins processing in parallel with a level of partitioned throughput. Streams can also trigger other instruction streams and multiple streams can synchronize with each other when necessary. For example, consider a machine running three streams concurrently and one of the streams halts. The other streams are automatically allocated the instruction slots which would otherwise be wasted.

Real-time systems also require hard deadline management which is often implemented via timer based interrupts. In conventional architectures, these interrupts require context switches. In DISC, an interrupt, instead of suspending a running process, can create its own instruction stream. When the interrupt routine finishes, the throughput will be dynamically reallocated to the remaining instruction streams. Context switching is not required as long as the number of instruction streams supported by the processor is less than or equal to that required by the application.

Other examples are the multiple instruction stream processor of Kaminsky and Davidson [kami79,kowa85], DART [shet91], and others [stal86,agar90,mccr91]. A dynamic interleaving model has been proposed to improve the rate of instruction issue to functional units [pras91]. Several researchers have proposed multithreaded superscalar architectures [dadd91,dodd92,pras91, hira92].

### 2.1.3 Superscalar Machines

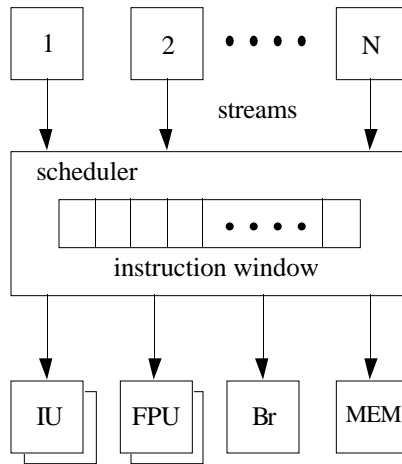
Even though the CDC 6600 [thor64] had multiple functional units, the idea of multiple instruction issue was not pursued until many years after its introduction in 1964. Several studies in the early 70's concluded that the instruction-level parallelism was very small [flyn70]. However, by the mid 80's, new work showed significant parallelism available.

Both Superscalar and VLIW machines exploit instruction-level parallelism of a single instruction stream. The primary difference is that superscalar processors perform run-time scheduling while in VLIW processors this is performed at compile-time [fish81, fish84]. A disadvantage of compile-time scheduling is the variable latency of memory operations; a data cache miss may stop issue of all subsequent instructions until the memory delivers the data, since most VLIW machines do not have data conflict resolution hardware. Dynamic scheduling in a superscalar processor does not have this problem. However, dynamic scheduling needs complex hardware for conflict resolution.

Instruction-level parallelism is defined as the average number of instructions issued per cycle in an ideal machine having an unlimited number of functional units and an infinite window size. The amount of parallelism is limited by the data, control, and structural hazards [henn90]. Several compiler techniques increase the amount of parallelism, for example, loop unfolding, software pipelining, and instruction reordering [henn90]. At the hardware level, techniques such as speculative execution and register renaming reduce but do not eliminate the effect of the hazards.

The performance gains possible through speeding up serial execution rates are very limited using reasonable hardware. Jouppi shows that the available parallelism for most nonscientific applications is around 2.5 instructions [joup89]. Results indicate that the parallelism in a program can vary by orders of magnitude over relatively short time intervals. Optimized numerical programs also show wide and rapid variance in available parallelism. Good machine utilization requires finding a way to smooth out these resource profiles.

Recent studies on the limits of instruction parallelism in applications show respectable speedups for many programs [aust90, butl92, theo92]. However, unrealistic assumptions such as big window sizes, perfect speculative execution, perfect register renaming, and memory disambiguation are needed to exploit that parallelism. For example, the study by Theobald et al [theo92] shows results for window sizes of 64 and 2048 instructions. The complexity of implementing a large window size is prohibitively expensive in both clock speed and chip area. Dwyer et al [dwye87] estimate that the issue, compression, and allocation logic for a proposed implementation of an 8-entry window based on the dispatch stack idea consumes 150,000 transistors. The complexity of a window is roughly proportional to the square of the window size since each instruction must check dependencies with all the instructions that precede it. We can see that big window sizes are out of reach of today's technology.



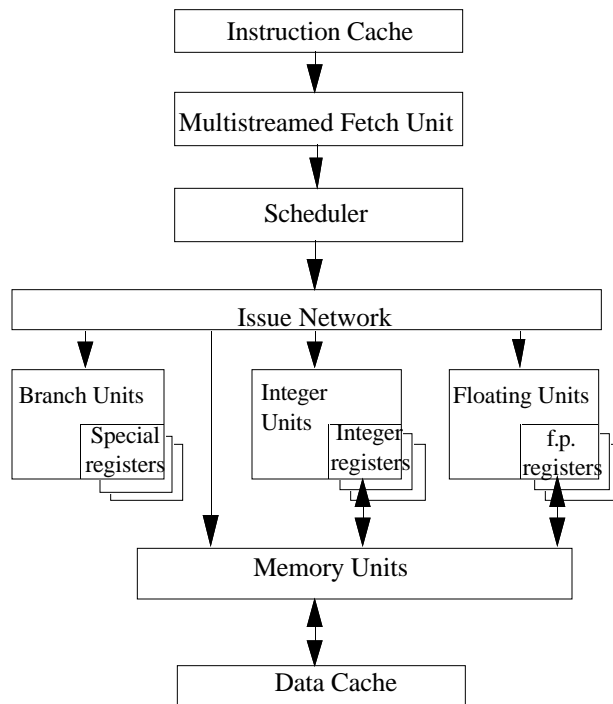
**Figure 2.3.** Block diagram of a multistreamed, superscalar processor.

## 2.2 Dynamic Multistreamed Superscalar Processors

A multistreamed superscalar processor is a multithreaded architecture that uses chip real estate efficiently to implement a processor that takes advantage of the multithreaded parallelism in applications. A multistreamed, superscalar processor is organized to support the simultaneous execution of several instruction streams by holding the context of each stream in hardware and fine-grain sharing of the functional units (cycle-by-cycle). Each hardware stream can be viewed as a *logical* superscalar processor.

Dynamic Multistreamed Superscalar Processors (DMSP) run multiple instruction streams simultaneously. DMSP processors combine the multiple instruction issue per cycle concept (superscalar) with the ability to dynamically interleave the execution of multiple threads. There are multiple active streams and the sequencer selects instructions from any of the active streams. The processor can adapt to the number of threads running. It can execute in single stream or multiple stream mode. When a single stream is running, the processor issues as many instructions as dependencies and the configuration allow. As the number of streams increases, the instruction issue scheme changes to accommodate instructions from several streams at the same time.

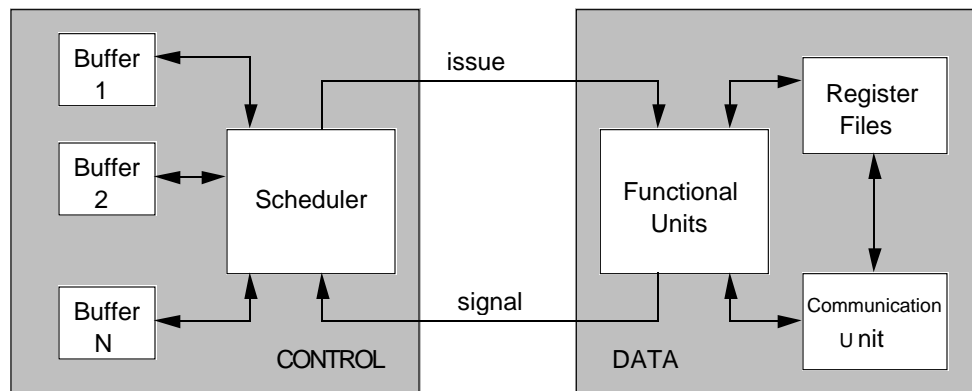
Figure 2.3 shows a simplified block diagram of the multistreamed superscalar processor. The diagram has three major parts: the instruction streams, the scheduler, and the functional units. An instruction stream consists of the context of a process and a buffer containing the



**Figure 2.4.** Detailed hardware organization.

next instructions to be executed. The scheduler selects instructions from the *stream issue windows*. Instructions within the stream issue window can be dispatched out-of-sequence provided no dependencies exist. The scheduler checks the instructions in each stream's issue window for data and control hazards and then moves the unblocked instructions into the global issue window. The global issue window contains all the instructions in the stream issue windows that are ready to execute, i.e., that have no data or control dependencies. From the global issue window, instructions are dispatched to the appropriate functional units provided no structural hazards are present. The processor has the capacity to execute  $N$  instruction streams simultaneously. In Figure 2.3, the streams share the functional units and multiple copies of certain functional units can improve overall performance as the number of streams increases. In Figure 2.3, the implementation has four functional unit types: two integer units (IU), two floating point units (FPU), one memory unit (Mem), and one branch unit (BR).

Figure 2.4 shows a more realistic hardware organization. The hardware is organized to support several instruction streams running concurrently. Each stream has its own program



**Figure 2.5.** Decoupling control and data.

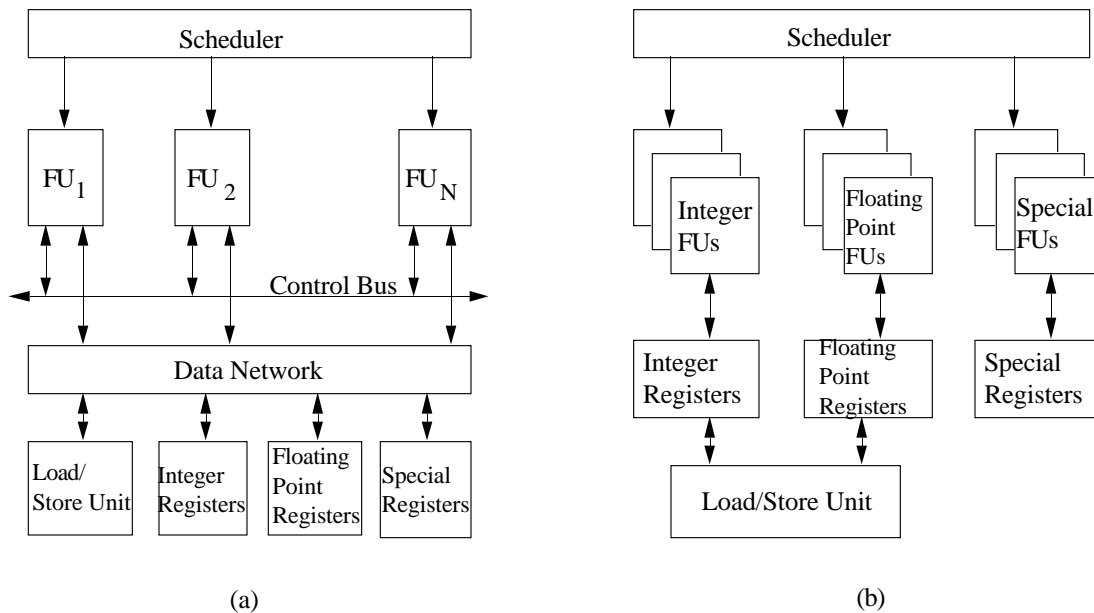
counter and register set. Instructions are brought by the fetch unit from the instruction cache and stored in each of the stream prefetch buffers. From these buffers, instructions are sent to the scheduling unit where the logic determines the data dependencies on its preceding instructions that have not yet completed. The scheduler dispatches instructions that are free of dependencies to the appropriate functional unit provided structural hazards do not exist. Instructions are dispatched to the functional units through an issue network.

Functional units return results to the corresponding stream's register file based upon a stream identification tag appended to each instruction. Also, functional units return instruction identification tags to the scheduler. As a result, the scheduler can release new instructions to the functional units. Interleaving requires additional hardware; however, many resources can be shared.

### 2.2.1 Decoupling Control and Data

To minimize global interconnections, data values and control signals are separated in the multistreamed processor, using the argument-fetching principle [denn88, mont92]. An instruction operates on values stored in registers and gets executed by a functional unit. Upon completion, the unit generates a signal to the control unit.

As seen in Figure 2.5, the scheduler takes instructions from the stream instruction buffers and issues them to the functional units. Instructions are sent to the scheduling unit where the logic determines the data dependencies on earlier uncompleted instructions from the same stream. The scheduler issues instructions free of conflicts to the functional units. The scheduler unit maintains register dependency tags. Functional units return the results to



**Figure 2.6.** Two approaches for the data network: a) using a global data interconnection network, and b) using the principle of data type locality.

the corresponding register file. The communication unit directly takes or deposits data from the register files to the external world.

The main characteristic of this model is that data never flows through the control unit. Instead, signals that hold the sequencing information among the instructions are the only entities that move around the circular structure of the processor [denn88]. The advantage of this approach is the reduction of global communication and the relaxing of the synchronization constraints between the different modules of the processor. This makes possible designs with a higher clock rate. Also, this reduces the complexity of the instruction issue network since only instructions are passed through, not data.

### 2.2.2 Data Type Locality Principle

Using the data type locality principle to reduce global interconnections, instructions are separated according to the data types they operate on. An instruction gets executed by a determined functional unit that has access to register files containing a data type. The communication between functional unit types processing the same data type is high, while the communication of functional unit types operating on different data types is kept to the minimum. Functional units are in proximity to their data type register stores.

Figure 2.6 (a) shows a common data network used to access different data types. Unfortunately, this is a complex network because a) the functional units must coordinate the use of the data network using a control bus, b) the data network must support many data types and many simultaneous accesses by the functional units, and c) the load/store unit must use the data network. The resulting data network employs a lot of global interconnection.

Under the principle of data type locality shown in Figure 2.6 (b), the functional units and their data types stores (registers) are in proximity. The functional units access their data types locally. Only in some cases it is necessary to provide global interconnection; for example, load/store units accessing the register files.

The Power architecture (see Figure 3.1, next chapter) uses this principle of data type locality [whit93]. The Power2 processor is separated in three integrated circuits: The ICU unit is in charge of prefetching and storing instructions in buffers; it also processes branches and dispatches instructions to the FPU and FXU chips; the special registers are contained in this chip. The FXU contains the integer register files and processes integer instructions using two functional units. The FPU contains two floating-point functional units that access the floating-point register files. Only for floating-point memory operations must there be synchronization between the FPU and FXU, because the effective address is contained in integer registers and the data in the floating-point registers. Also, branch operations need communication since operands are compared in either the FXU or FPU for integer or floating-point operands, respectively. The flags resulting from the comparison are transferred to the ICU for branch processing. Using this locality principle the Power architecture separates the processor in integrated circuits with the minimum of communication between them. It also helps to provide some decoupling between the integrated circuits.

### 2.2.3 Scheduling Instructions

The number of instructions issued by single thread is restricted by the data and control dependencies between instructions. However, instructions from different independent threads can be issued concurrently. One of the objectives of a multistreamed architecture is to schedule instructions from alternate threads whenever conflicts lead to wasted cycles in a thread. In every clock cycle the scheduler interleaves instructions from different threads to keep the utilization of the functional units at their highest.

One important objective is to provide excellent single thread performance, i.e., when only a single thread is available the processor should be as fast as an equivalent conventional superscalar processor. Thus, the processor could dynamically work in either single-stream or multiple-stream mode, according to the number of streams available. When the processor is working in a single-stream mode, the issue scheme is vertical by taking instructions from the same stream, as in a conventional processor. In a multiple stream mode the issue scheme adds a horizontal component by taking instructions from different streams.

Several authors have proposed static schemes for scheduling [wang91]. In a static scheme, the scheduler extracts information at compile-time indicating the precise schedule to be followed at run-time. Static scheduling works well for the single thread if the processing elements are tightly synchronized; the scheduler knows precisely how many cycles an instruction will take (except for memory and branch instructions). While we do not discard that static scheduling is useful to schedule instructions for some multithreaded programs (for example, threads extracted from an iteration loop), we feel that dynamic hardware scheduling at run-time makes better use of the multistreamed processor in more cases than the static scheme.

Dynamic scheduling allows several streams to issue instructions whenever the single stream does not have enough ready-to-issue instructions for the functional units. Two (often conflicting) objectives for a dynamic scheduler are: a) maximize the overall efficiency of the processor, and b) maximize the effective processor share given to each thread (responsiveness), important in real-time systems to guarantee a minimum of processor throughput to each thread.

A dynamic schedule to maximize the overall efficiency could select threads using a priority scheme and a round-robin strategy. Figure 2.7 shows an example with three instruction threads and four functional units: two integer (I), one float (F), and one load/store (L). We assume that instructions execute in a single cycle, and that the memory is perfectly pipelined. Figure 2.7 (a) shows the ready-to-issue instructions from threads 1, 2, and 3 at clock=1; thread 1 has two ready-to-issue instructions while threads 2 and 3 each has three. The instruction windows can contain a maximum of three ready-to-issue instructions. The notation  $I_{i,j}$  means the  $i$ th instruction (integer) from thread  $j$ ; thus  $I_{1,1}$  is an integer instruction that occurs just before the floating-point instruction  $F_{2,1}$  in thread 1. The notation (-) indicates an empty slot, i.e. there is no a ready-to-issue instruction in this space.

thread 1	thread 2	thread 3
-	* $L_{3,2}$	$L_{3,3}$
* $F_{2,1}$	$I_{2,2}$	$F_{2,3}$
* $I_{1,1}$	* $I_{1,2}$	$I_{1,3}$

(a) clock=1

thread 1	thread 2	thread 3
-	-	* $L_{3,3}$
$L_{4,1}$	* $I_{4,2}$	* $F_{2,3}$
$I_{3,1}$	* $I_{2,2}$	$I_{1,3}$

(b) clock=2

thread 1	thread 2	thread 3
-	$I_{7,2}$	-
$I_{4,1}$	$I_{6,2}$	* $F_{4,3}$
* $I_{3,1}$	* $L_{5,2}$	* $I_{1,3}$

(c) clock=3

Clock	Integer 1	Integer 2	Float	L/S
1	$I_{1,1}$	$I_{1,2}$	$F_{2,1}$	$L_{3,2}$
2	$I_{2,2}$	$I_{4,2}$	$F_{2,3}$	$L_{3,3}$
3	$I_{1,3}$	$I_{3,1}$	$F_{4,3}$	$L_{5,2}$

(d) summary of issue to functional units

**Figure 2.7.** Example of dynamic scheduling instructions from different threads using round-robin. Instructions issued in each cycle are marked with (\*).

At clock=1 (a), thread 1 is selected as the highest priority; thread 1 issues one integer ( $I_{1,1}$ ) and one float ( $F_{2,1}$ ). There are still one integer and load/store free functional units, so thread 2 is selected on a round-basis, and issues  $I_{1,2}$  and  $L_{3,2}$ .

As a result of issuing instructions at clock=1, new instructions become ready-to-issue in the instruction windows of threads 1 and 2 ( $I_{3,1}$ ,  $L_{4,1}$ ,  $I_{4,2}$ ). At clock=2 (b), thread 2 is selected as the highest priority and issues  $I_{2,2}$  and  $I_{4,2}$ . The next in the priority line is thread 3, which issues  $F_{2,3}$  and  $L_{3,3}$ .

Thread 3 becomes the highest priority at clock=3 (c) and issues  $I_{1,3}$  and  $F_{4,3}$ . Next, thread 1 issues  $I_{3,1}$  and since it does not have any load/store instructions, thread 2 issues  $L_{5,2}$ . Figure 2.7 (d) shows the summary of instructions issued in these three clock cycles.

The processor utilization is maximal for our example. This schedule does not always obtain the maximum efficiency, but it provides a degree of 'fairness' for the threads. This schedule can be modified to maximize the processor share given to a thread by fixing the priority of the streams. For example, if thread one always has the highest priority, its performance is equivalent to the single thread one. If thread two is next in the priority line, it obtains the 'leftover's from thread one. Thread three would issue only to those functional units not used by threads one and two. In any case, Chapters 3 and 4 analyze the characteristics of several schedules under determined workloads ●

## 3 Performance Evaluation of Multistreamed Architectures

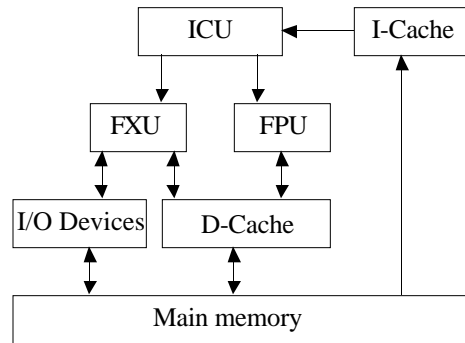
A way to evaluate performance is to build a hardware prototype, involving high development cost. Instead, we use a software simulator to evaluate different configurations at a fraction of what a prototype would cost. The idea is that the simulator interprets the instruction set of some existing architecture and simulates the scheduling of instructions on a new architecture. In this way, an existing base of programs and utilities can be used in testing a new architecture, reducing the costs and time of developing. The simulator is used to study the effect of different hardware configurations on the performance. In addition, the simulator helps to identify bottlenecks in designs and perform corrective measures.

### 3.1 A Multistreamed Simulator for the Power Architecture

#### 3.1.1 The Power Architecture

We chose the IBM Power architecture [groh90,hall91,ibm92,moto93,oehl91,whit93] as the base for our multistreamed architecture. Implementations of the Power architecture can issue more than one instruction per cycle, and support a limited out-of-order execution. The Power architecture has separate pipelined functional units that execute most instructions in a single cycle: fixed point units (FXU), floating point units (FPU), and a branch unit (ICU). Each unit type is located on a different chip and has its own set of registers. Figure 3.1 shows the interaction among the three units, the instruction and data caches (I-Cache, D-Cache), and main memory.

The instruction set defines 184 instructions that are executed by one of three functional unit types. A few of the instructions are executed on one unit type, but alter or use registers located on another unit type. For example, the floating-point compare is executed in a floating-point unit and alters the condition register (CR) located on the branch unit.



**Figure 3.1.** Functional blocks of the Power architecture.

### 3.1.2 Powersim

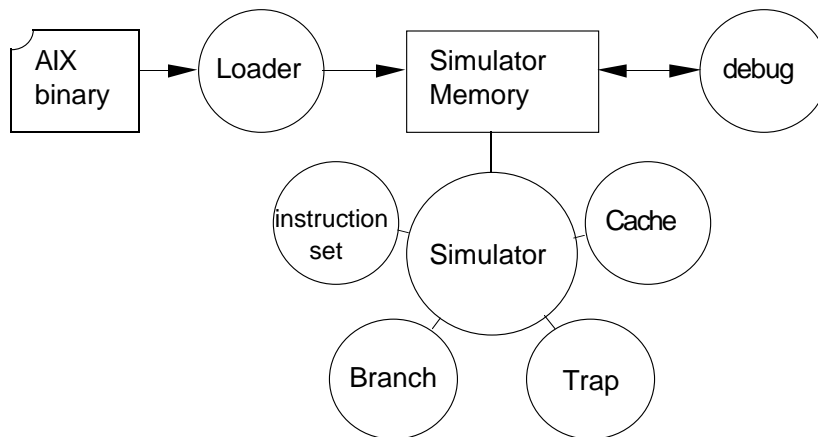
Powersim is the interactive program that we developed to simulate the operation of a multithreaded Power computer. Powersim can load programs compiled for the AIX operating system, interpret the instruction set of the RS/6000, and simulate the issue of instructions to different functional units in a multistreamed architecture.

load	load AIX executable binaries.
run, step	run or single step instructions.
get, put	manipulate data from registers and memory.
stop, trace	debug (breakpoints, trace).
stats	get diverse statistics about program execution.

**Table 3.1.** Principal commands of Powersim

Powersim provides an interactive programming environment. Commands are provided for debugging, single-stepping, continuous execution, view/modify internal data, and collect statistics of the execution of the program. The user can collect information on the run-time properties of a program, for example, the utilization of the functional units and the performance of each stream under a determined hardware configuration. Table 3.1 summarizes the principal commands of Powersim.

Our simulator is based on the DLX simulator [henn90]. We rewrote most of the code to provide support for the Power architecture, superscalar, multistreaming, and cache. Our objective was also to build a simulator that could be targeted for diverse architectures.



**Figure 3.2.** Functional block diagram of Powersim.

Figure 3.2 shows the division of modules in Powersim. The *loader* interprets the header portion of an AIX executable binary and loads the program into the simulator memory. The *simulator* is the part of the code that interprets the instruction set and schedules operations to the functional units. The *C-Trap* handler module emulates the execution of C-library routines that a program calls. Memory references are passed to the *cache* simulator module. The *branch* module implements different branch prediction schemes. The *debug* module handles breakpoints and disassemble instructions when executed. The functionality provided by *debug* is similar to the standard Unix utility *dbx*.

The modules are interconnected through the *Tcl* language interface [oust93], which provides the interface to the user. *Tcl* is an interpreted application-independent command language. It consists of a library package that is embedded in tools (such as editors, debuggers, etc.) as the basic command interpreter. *Tcl* provides a parser for a simple textual command language, a collection of built-in utility commands, and a C interface that tools use to augment the built-in commands with tool-specific commands.

### 3.1.3 A High Level Model of Instruction Execution

In order for the simulator to be useful in running realistic workloads, the simulation needs to be *fast*. Therefore, a detailed hardware description is not implemented in Powersim. Instead, Powersim is a functional simulator using a high level model of instruction execution reflecting the hardware behavior. Clearly, there are differences in the performance measures of Powersim and the real hardware; however the differences in run-times of programs is

small. For example, the average error for the floating-point SPEC'92 benchmark set running in the original RS/6000 is 10%, with a peak error of 20% occurring in only one of the benchmarks.

The instruction execution model specifies a  $(unit, latency, lock)$  for each instruction. The *unit* is the functional unit that will execute the instruction. The *latency* is the number of cycles that the functional unit will take to produce the result. The *lock* is the number of cycles that the functional unit will be unavailable while executing the instruction. For example, perfectly pipelined units have a  $lock=1$ . Another example is the instruction *div* (integer divide) which uses the integer unit, takes 17 cycles to produce a result, and locks the unit while executing, i.e.,  $div=(unit, latency, lock)=(integer, 17, 17)$ .

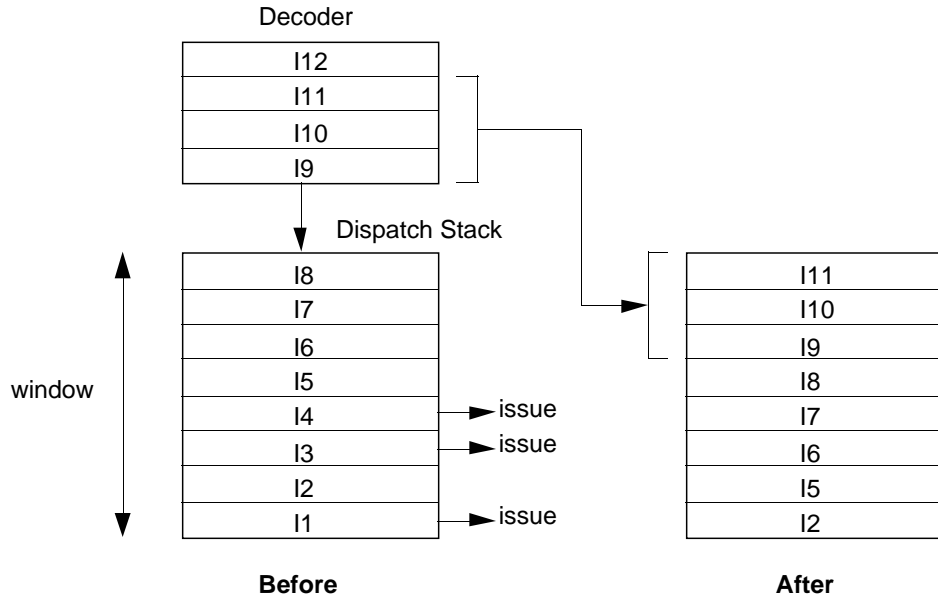
In addition to the basic functional unit *latency*, some instructions may add a *run-time* latency. For example, load/store instructions add a cache access latency that depends on the cache behavior. Another example is branch instructions, which may add a performance-penalty latency in case that branches are mispredicted. The Power architecture specifies the fall-through path of branches as the predicted path, with software hints to change the prediction [moto93].

### 3.1.4 Hardware Simulation in Powersim

Powersim simulates diverse hardware modules, including instruction and data caches, instruction buffers, scheduler, and functional units. We describe the hardware simulation scheme of Powersim.

Instructions are fetched from the I-Cache and stored in issue buffers. An instruction must pass the data (RAW,WAW,WAR), control, and structural tests before it issues to one of the functional units. An instruction stalls in the issue buffer if any of these tests is not passed. An instruction must pass data and control checks with all other instructions preceding it in the issue buffer and in execution in the functional units. Even if the data and control tests are passed, an instruction must check the availability of a functional unit (structural test).

Powersim marks instructions that have passed the data and control dependencies checks as ready-to-issue. Powersim's scheduler may dispatch instructions out-of-order within the stream issue window in the absence of data hazards. An instruction is removed from the stream issue window as soon as the scheduler dispatches it to a functional unit. The window is then compacted and refilled with new instructions.



**Figure 3.3.** Operation of the Dispatch Stack.

Instructions are stored initially in a prefetch buffer. Instructions are then transferred to the dispatch stack [dwy91]. The dispatch stack performs operations that are very similar to those performed in a reservation station. Instructions are placed at the top of the dispatch stack by the decoder and are issued from the bottom of the stack. Instructions conserve the decoding order; the oldest instruction is at the bottom of the stack, with the highest priority. An instruction can remain in the dispatch stack for any arbitrary amount of time until it issues. The number of entries of the dispatch stack defines the instruction window. The operations performed in the window are:

- Identify ready-to-issue instructions by checking data and control dependencies.
- Select instructions based on structural requirements. If more than one instruction require the same functional unit, the one with the highest priority is selected. A functional unit accepts an instruction if the unit is free and there is a bus (if defined) connected to the register file, such that the operands can be fetched.
- Issue instructions to the selected functional units. Their respective slots are freed and the stack is compacted as shown in Figure 3.3. The registers that receive results are locked while the instruction is executing, preventing succeeding instructions from using them.

The multistreaming operation is similar to the single stream operation discussed so far. The difference is that streams are assigned different priorities to issue. At every cycle, one stream is selected as the highest priority, from a priority list specified by the user. Instructions that pass all the checkings are selected from the highest priority stream and sent to the functional units. If there are free functional units after the highest priority stream has issued, the next stream is selected from a *give away* list of streams. Issue stops in a determined cycle when either the give away list is exhausted or all the functional units are busy. The *priority* and *give away* lists are advanced in a cyclic manner, similar to the operation of a circular shift register. For example, if the priority list is  $(1,2,3)$ , then the highest priority stream at consecutive cycles is: 1,2,3,1,2,3,1,...,etc.

Powersim allows the user to specify many parameters for the different hardware modules. For example, the scheme used for branch prediction can be chosen. The user can select the functional unit types and the latency of each instruction. Table 3.2 summarizes the different parameters that can be specified for the simulation.

Hardware module	Description
• Icache, Dcache	Specification parameters of the instruction and data caches.
• Issue buffer	Window of instructions and size of the buffer
• Branch unit	Branch prediction scheme and misprediction penalty.
• Scheduler	Lists used by the scheduler (type, priority, and give away).
• Functional units	Type, number, and latency of functional units.
• Buses	Communication buses for units, register files, and caches.

**Table 3.2.** Powersim's hardware specification parameters.

### 3.2 Preliminary Study

We present a preliminary study of our multistreaming architecture, using Powersim and the SPEC'89 benchmark set. The following is our simulation framework and its justification:

- Fully pipelined functional units were assumed. Using non pipelined FUs will show greater benefit of multithreading since the cost of data dependencies increases. Instructions execute in exactly one of the following functional units: Integer,

Floating Point, Multiply (Integer), Divide (Integer), CR (conditional register), Branch, and Memory (Load/Store Unit).

- Register renaming is not used. Register renaming can increase the performance on the single streamed processor; however, we do not include renaming to show the performance advantage of multistreaming.
- Branches do not incur any performance penalty. No extra cost for branches was assumed to obtain a more conservative result. As the cost of a branch increases, the performance of a single-streamed architecture reduces but its penalty is much lower in a multistreamed architecture.
- Single-cycle memory access is used to exclude the effect of memory/cache and concentrate on data, control, and structural dependencies. As the memory latency increases, the benefit of multistreaming increases too, since there are other streams ready to execute. Therefore, this assumption produces a conservative result. Later, it is shown that a larger latency will result in better performance of multistreaming over single stream.
- We compiled the SPEC'89 benchmarks with the C optimizer (-O) option. The AT&T Fortran-to-C converter [feld91] translated the Fortran benchmarks. No modifications to the code were performed to increase the performance. The benchmarks were not run to completion and a statistical sampling method was used [cont92] to reduce the simulation time. However, each simulation was run with at least one hundred million (100 M) simulated cycles.

### 3.2.1 Effect of Configuration

We examine the effect of the configuration on the performance. The machine configurations match functional unit capabilities to instruction class frequencies; frequencies for the benchmarks used are shown in Table 3.3. Table 3.4 lists the number of functional units for configurations *C1*, *C2*, and *C3*, and the latency of each functional type. Configuration *C1* resembles closely (except for multiply and divide units) the original RS/6000 architecture. Configurations *C2* and *C3* represent a step further in exploiting higher performance. The *integer*, *float*, and *memory* types have more functional units, as they appear more frequently. The *multiply* (integer), *divide* (integer), and *cr logic* (conditional register) frequencies are too low to justify more than one unit.

Benchmark	integer	float	multiply	divide	cr logic	branch	memory
eqntott	39.3	0.0	0.0	0.0	0.0	35.5	25.2
espresso	51.5	0.0	0.1	0.0	0.7	21.9	25.8
li	36.4	0.0	0.0	0.0	1.3	21.2	41.0
doduc	17.3	27.8	0.2	0.0	3.3	10.2	41.2
fpppp	3.6	35.3	0.0	0.0	0.3	2.3	58.4
matrix300	1.4	16.2	0.1	0.0	0.0	16.7	65.5
spice2g6	43.5	1.9	0.3	0.0	0.7	19.6	33.8
tomcatv	6.1	45.1	0.0	0.0	3.4	9.3	36.1

**Table 3.3.** SPEC '89 instruction mixes. The numbers indicate the percentage of each instruction type.

Functional unit	Latency	Configuration		
		<i>C1</i>	<i>C2</i>	<i>C3</i>
integer	1	1	2	6
float	2	1	2	3
multiply	5	1	1	1
divide	17	1	1	1
CR logic	1	1	1	1
branch	1	1	1	2
load/store	1	1	2	3

**Table 3.4.** Configurations evaluated.

We obtained results with the following simulation framework:

- Most of the simulations use a window size of four for each stream; however configuration *C2* uses a window size of 8 when running one stream, and *C3* uses a window size of 16 and 8 for one and two streams, respectively. This is necessary to adequately exploit *C2* and *C3*, since they have a total of 10 and 17 functional units, respectively.
- Multiple copies of the same program were loaded in all the streams of the machine. The copies share the same code segment, but have independent data segments. This is a worst-case since the probability of stream inter-collision is higher.
- A *fair* schedule is used. In any given cycle, the highest priority stream is chosen randomly among all the active streams. The highest priority stream issues as many

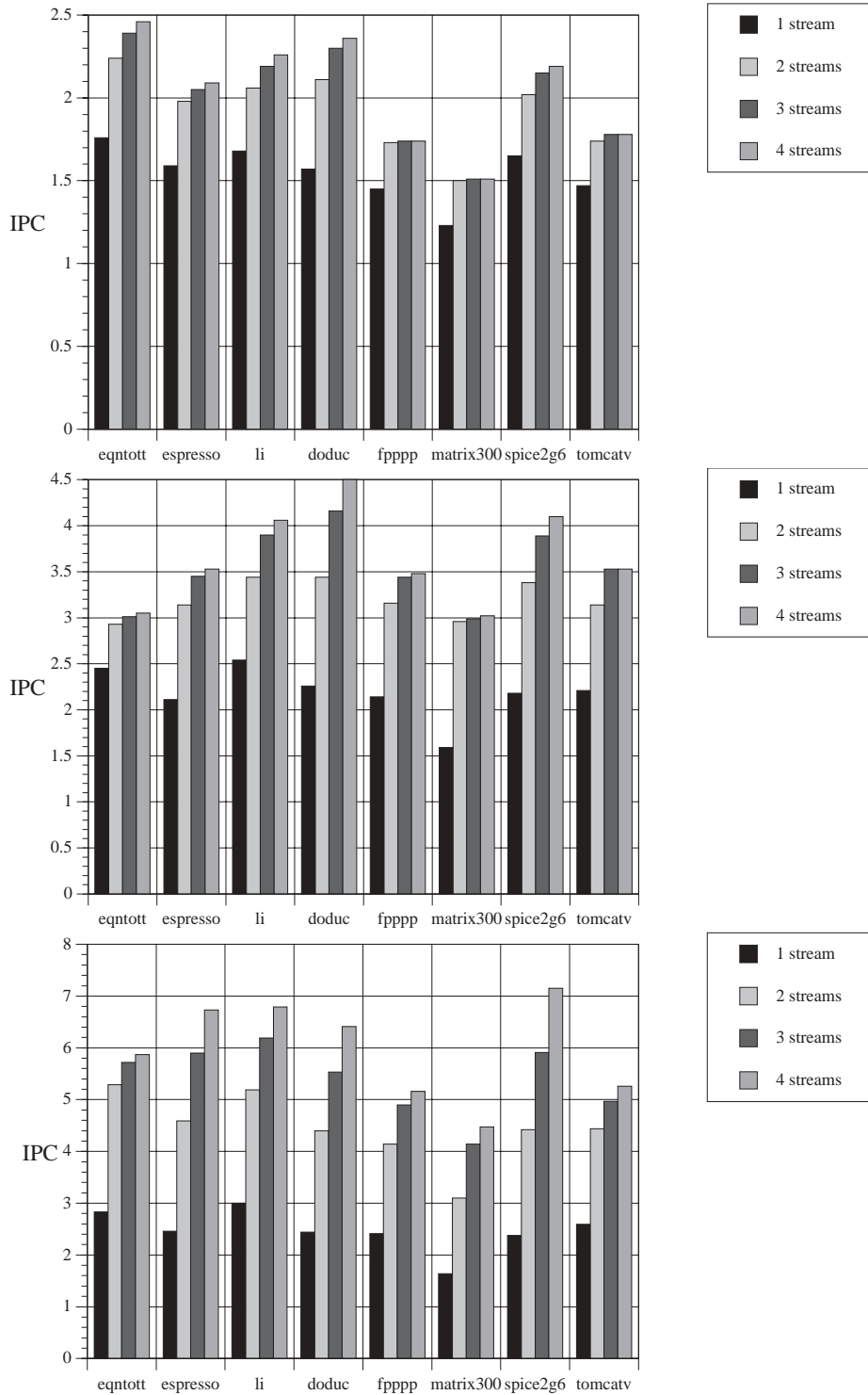
instructions as it can in that cycle. If 'free' slots in functional units exist, then the other streams issue instructions to these 'leftover' functional units. The scheduler is 'fair' since each stream is given the highest priority equal number of times.

Figure 3.4 shows the results of the simulations. The columns indicate the Instruction-per-Cycle (IPC) for one, two, three, and four streams. One can notice that for configuration *C1*, small increases in IPC are obtained as the number of streams is increased. This effect is due to saturation.

Saturation depends on the machine configuration. Saturation is reached when the utilization of one functional unit type reaches one hundred percent; the functional unit becomes the bottleneck of the system. For example, if integer operations are the majority of the instructions and the machine has only one integer unit, then the integer unit is likely to become saturated and, therefore, multistreaming does not increase performance. Thus, the contention for the integer unit primarily determines execution time.

As a first order approximation, the IPC of saturation can be determined by the minimum of the number/frequency ratios, or  $\min_{unit=1}^T c_{unit}/v_{unit}$ , where  $T$  is the number of functional unit types,  $\mathbf{V}=(v_1, v_2, \dots, v_T)$  is the instruction mix vector (i.e. the fraction of each instruction type), and  $\mathbf{C}=(c_1, c_2, \dots, c_T)$  describes the number of functional units of each type. For example, memory operations dominate *matrix300* (65.5%); the saturation point for configurations *C1*, *C2*, and *C3* are given by  $1/0.655=1.53$ ,  $2/0.655=3.05$ , and  $3/0.655=4.58$  respectively. These saturation values match closely the simulation results.

Most of the benchmarks reached saturation for configuration *C1* with as few as two streams. The instruction-level parallelism of the benchmark is sufficient to adequately exploit the configuration. The improvement for configuration *C1* with four streams is low because one stream can almost reach saturation; the improvement is approximately 30%. Configurations *C2* and *C3* obtain larger throughput gains. As the number of streams increases, performance grows to exploit the instruction parallelism. The saturation point determines the upper bound of performance. Increasing the number of functional units increases the performance for one stream until data and control dependencies dominate performance.



**Figure 3.4.** Statistics for configuration *C1* (top), *C2* (middle), and *C3* (bottom), with random (fair) schedule. The columns indicate the Instruction-per-Cycle (IPC) for one, two, three, and four streams.

Number of streams	Configuration		
	C1	C2	C3
1	0	40.8	58.8
2	23.8	108.0	185.5
3	29.3	130.0	248.1
4	31.5	137.0	284.6

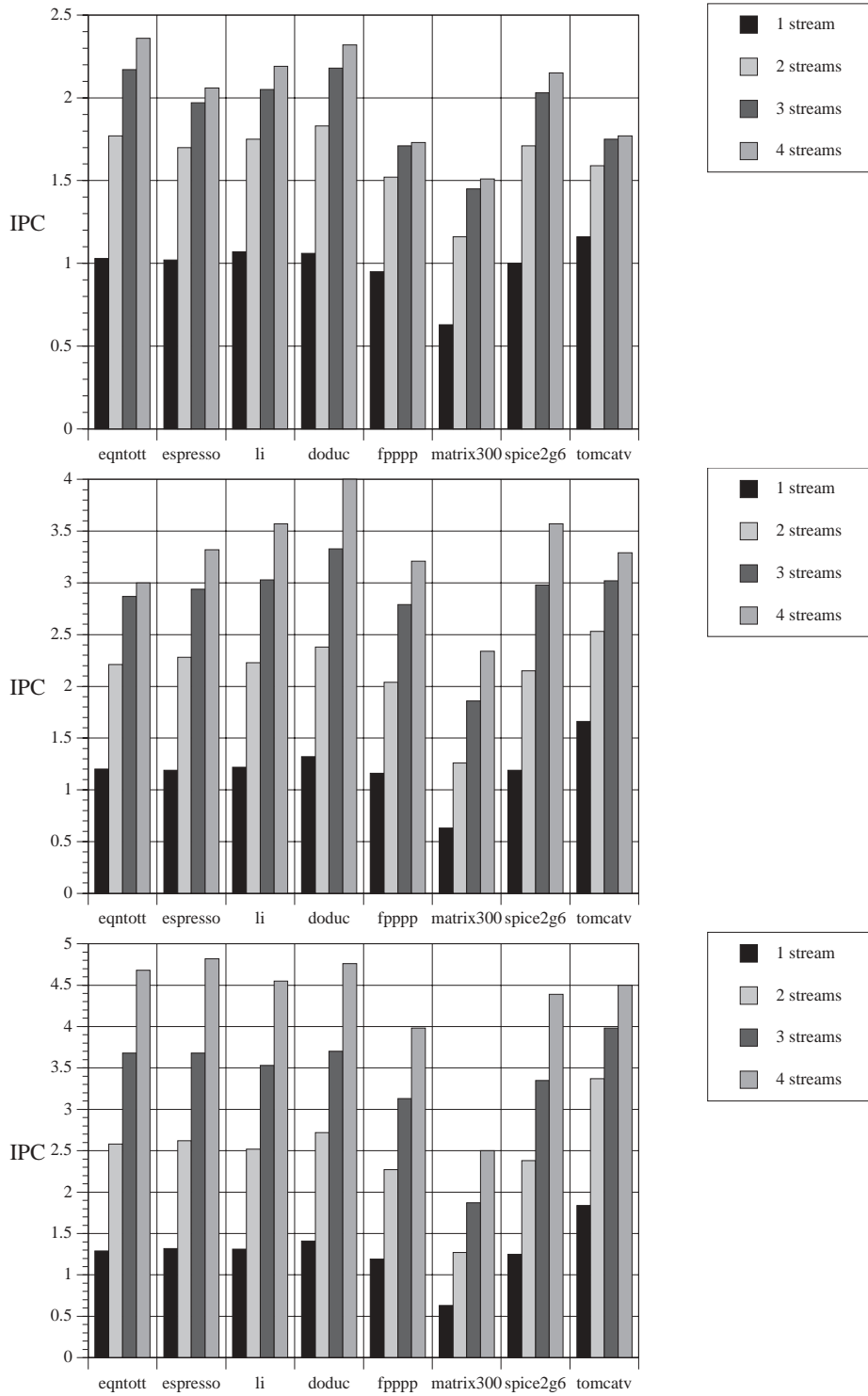
**Table 3.5.** Multistream improvement with a random schedule. Average percentage improvement compared to configuration C1, single-stream.

Table 3.5 shows the average gain from multistreaming. The reference point is the performance for one stream in configuration *C1*. Performance grows with the increase in the number of functional units up to 59% for configuration *C3*. However, resource usage is lower since configurations *C3* has 17 functional units compared with 7 for configuration *C1*. Adding more streams improves performance (up to 285% for four streams) resulting in better resource usage.

### 3.2.2 Effect of Memory Latency

Section 3.2.1 assumed single memory latency. We now examine simulations with a memory latency of four. Memory is still assumed perfectly pipelined, and instruction fetch is single cycle. In our scheme, the stream remains active, while the load/store functional unit performs the memory operation. In the meantime, ready-to-issue instructions from the same stream can issue. The main purpose of this test is to demonstrate that the benefit of multithreading increases as the latency increases.

Figure 3.5 show the results. The higher memory latency resulted in lower single stream performance, as expected; however, the improvement of multistreaming is more evident. Even configuration *C1* shows significant improvement from multistreaming. The higher the percentage of memory instructions in the benchmark, the higher the improvement. For example, *matrix300* is a memory-bound benchmark (65.5%) that exhibits the highest improvement with multistreaming. Table 3.6 shows the multistreaming improvement for configurations *C1*, *C2* and *C3*. The reference point is the performance of one stream for each configuration. The upper bound of improvement is 100% for each additional stream. This improvement is achieved in *matrix300*; the performance grows linearly with the number of streams.



**Figure 3.5.** Statistics for configuration *C1* (top), *C2* (middle), and *C3* (bottom) with random (fair) schedule and memory latency 4.

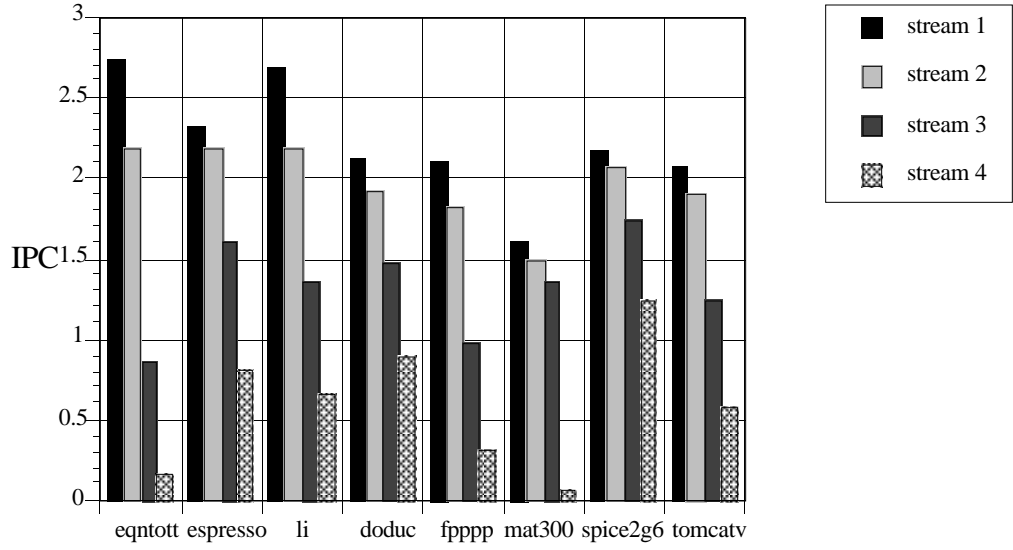
benchmark	configuration C1			configuration C2			configuration C3		
	2	3	4	2	3	4	2	3	4
eqntott	71.8	110.7	129.1	84.2	139.2	150.0	100.0	185.3	262.8
espresso	66.7	93.1	102.0	91.6	147.1	179.0	98.5	178.8	265.2
li	63.6	91.6	104.7	82.8	148.4	192.6	92.4	169.5	247.3
doduc	72.6	105.7	118.9	80.3	152.3	203.0	92.9	162.4	237.6
fpppp	60.0	80.0	82.1	75.9	140.5	176.7	90.8	163.0	234.5
matrix300	84.1	130.2	139.7	100.0	195.2	271.4	100.0	196.8	296.8
spice2g6	71.0	103.0	115.0	80.7	150.4	200.0	90.4	168.0	251.2
tomcatv	37.1	50.9	52.6	52.4	81.9	98.2	83.2	116.3	144.6
AVERAGE	65.9	95.6	105.5	81.0	144.4	183.9	93.7	167.5	242.5

**Table 3.6.** Multistreaming improvement for memory latency 4, random schedule and configurations C1, C2, C3. Percentage of improvement for 2, 3, and 4 streams over one stream for each configuration.

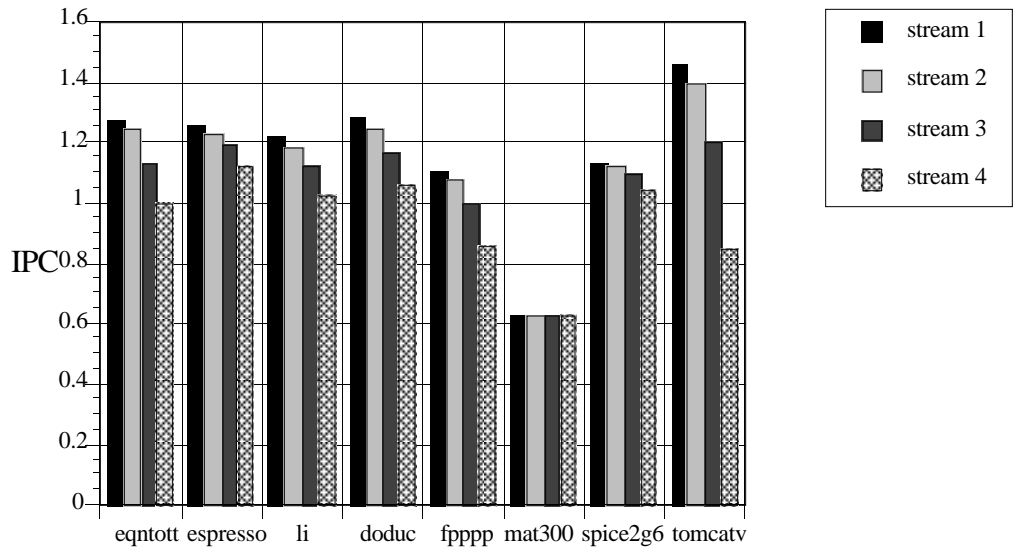
### 3.2.3 Effect of Schedule

The scheduler in a dynamic interleaved architecture adds overhead and complexity to the architecture. Different schedule schemes have diverse implementation costs. Here we compare the fair schedule used in Sections 3.2.1 and 3.2.2 with a *prioritized* (or unfair) schedule. In the prioritized schedule, the priority of the streams is fixed. Stream 1 has the highest priority and stream 4 the lowest priority. Stream 1 issues as many instructions as it can. Clearly, stream 1 has the same performance as it was running in a single streamed machine and stream 4 can only issue in a given cycle if there are remaining functional units after streams 1, 2, and 3 issue.

The simulations were run with all streams active throughout the run. The IPCs obtained from the two schedules are similar, as expected. This does not mean that the individual stream performance is similar; the prioritized schedule clearly makes some streams starve. More interesting is to observe the effect of the memory latency on performance for the prioritized schedule. For our experiment we chose configuration C3 and ran it with four streams, with memory latency 1 and 4. Figure 3.6 and Figure 3.7 show the results.



**Figure 3.6.** Performance of each stream for configuration C3, prioritized schedule, and memory latency of 1. IPC of each individual stream for a run with four streams.



**Figure 3.7.** Performance of each stream for configuration C3, prioritized schedule, and memory latency of 4. IPC of each individual stream for a run with four streams.

Figure 3.6 shows the performance of each stream for the prioritized schedule for memory latency 1. Stream 1 has the highest priority. Thus, stream 1 has the same performance as if it were running alone. The results for stream 1 are slightly lower than the ones presented in Figure 3.4 because we used a window size of 4 (previous window size was 16). Stream 2 is next in the line of priority. Stream 2 has almost 80% of the performance of stream 1 running in a cycle steal mode or more precisely in a functional unit steal mode. In other words, stream 2 will steal (or use) the functional units that stream 1 cannot use because of dependencies; thus, the configuration is adequate to support two streams. The performance of the third stream varies with the benchmark, for example *matrix300* and *spice* have enough leftover instructions to support this stream, while the performance for *eqntott* is regular. The fourth stream shows a significant decrease of performance.

Figure 3.7 shows the effect of increasing the memory latency to four. A high memory latency is an obstacle to exploiting instruction-level parallelism. Thus, increasing the memory latency levels the performance of the four streams. The figure shows that a single stream cannot exploit the configuration adequately. There are enough resources in configuration *C3* to support four streams. Thus, performance growth is almost linear with the number of streams. By using multistreaming, one virtually gets an extra processor with memory latency of 1. We see that this almost approximates the work of four single stream processors●

## 4 Performance Estimation of Multistreamed Architectures

We present a model to estimate the performance of superscalar multistreamed architectures.. The input parameters of the model are the workload and architectural descriptions that are estimated or obtained using commonly available tools. The model produces instructions executed per cycle (IPC) estimates. To validate this technique, estimates of the model for several benchmarks are compared against results from Powersim.

A simulation study demonstrating performance benefits of multistreamed, superscalar processors was presented in Chapter 3. Due to the cost of the simulations in this study, both in complexity and execution time, we develop a simple analytical model to estimate the overall performance of these architectures. A purely analytic model is often too complex. Often, many simplifying assumptions have to be taken for the model to be tractable. Instead, we propose a model that combines simulation and analysis to get performance measures faster than a pure simulation method. Our approach views the instruction stream as a random source of instruction types. The characteristics of the instruction stream are extracted from simulations and used in the analytic model. Our results show that this interpretation gives a good approximation in the context of independent instruction streams. In addition, the model allows us to analyze the behavior of the design using programs for which we know the characteristics but do not have the code.

### 4.1 The Model

The model we developed is stochastic in nature and is based upon a Markov model. The model calculates the expected overall performance in number of instructions executed per cycle (IPC) by the processor. The performance of individual streams is not calculated by the model since we are attempting to calculate the best possible overall system performance independent of individual stream performance. The overall performance uses this formula

$$IPC = \sum_{w=1}^G P_w \cdot IPC_w \quad (4.1)$$

where  $P_w$  is the probability of having  $w$  instructions in the global window,  $IPC_w$  is the expected IPC measured for an issue window of size  $w$ , and  $G$  is the size of the global window. The technique is separated into two major parts: one modeling structural hazards (contention for resources) and the other modeling control and data hazards (RAW, WAW, WAR, and branches).  $IPC_w$  models the effect of structural hazards while  $P_w$  is a scaling factor used to model the performance degradation due to the data and control hazards. The rationale behind this division is that the modeling of structural hazards is primarily dependent on the architectural (hardware) configuration while the modeling of control and data hazards is primarily dependent on the workload. In addition, since we employ a Markov chain in our technique, combining the structural hazards as well as dependencies into a single chain results in an extremely large number of states. Our division simplifies the Markov Chain and makes the problem more tractable.

#### 4.1.1 Architecture Characterization

The architecture of the processor is specified by a number of streams ( $N$ ), a stream issue window size ( $S$ ), and a functional unit configuration vector ( $\mathbf{C}$ ). The stream issue window is the buffer consisting of instructions that can be issued in a given cycle. Issued instructions are removed from the window and new instructions are brought in the buffer to maintain it full. We assume that all the stream windows are of the same size, i.e.,  $S_1 = S_2 = \dots = S_N = S$ . Thus, the total number of instructions in the machine ( $G$ ) considered for issue is equal to  $G=NS$ , i.e. the sum of all the stream windows.

An instruction is ready-to-issue if it passes the data and control dependency constraints. A ready-to-issue instruction is issued if there is any functional unit that can accept it in a given cycle. We assume that there is a perfect scheduler that dispatches instructions from the streams in a fair manner, i.e. giving a fair opportunity to each stream to issue instructions. Functional units are assumed perfectly pipelined so each unit can accept an instruction every cycle. The parameter  $T$  describes the number of distinct functional unit types. The functional unit configuration vector ( $\mathbf{C}$ ) describes the number of units of each type. Each element  $c_i$  of  $\mathbf{C}$  contains the number of functional units of type  $i$ . For example, the IBM RS/6000 can issue up to four instructions per cycle (integer, branch, floating point, and condition register). Its parameters are:  $S=4$ ,  $T=4$ ,  $\mathbf{C}=(c_{integer} \ c_{branch} \ c_{floating-point} \ c_{cr})=(1,1,1,1)$ . Table 4.1 summarizes the architectural specification parameters.

$N$	Total number of streams.
$S$	Stream issue window size.
$\mathbf{C} = (c_1, c_2, \dots, c_7)$	Functional unit configuration vector

**Table 4.1.** Architectural specification parameters.

### 4.1.2 Workload Characterization

The workload is the programs or threads that are running on the multistreamed processor. The workload is characterized by the type and number of threads loaded and running in the hardware streams of the machine. We assume that each stream is loaded with a thread; thus, the number of threads is equal to the number of streams.

The workload of the system is characterized by its runtime instruction mix vector ( $\mathbf{V}$ ), the number of active streams ( $N$ ), and a histogram vector ( $\mathbf{H}$ ). We view each thread as a stochastic stream of instructions of different types, and we use the instruction mix to characterize the thread. The instruction mix specifies the percentage of instructions of each type. For example, the instruction mix for the benchmark *hanoi* is: 56% integer, 12% branch, and 32% memory instructions, e.g.  $\mathbf{V} = (v_{integer}, v_{branch}, v_{memory}) = (0.56, 0.12, 0.32)$ . The histogram vector ( $\mathbf{H}$ ) models the data and control dependencies. A complete description of the histogram vector will be presented in the subsequent sections. Table 4.2 summarizes the workload characterization parameters.

$\mathbf{V} = (v_1, v_2, \dots, v_7)$	Runtime instruction mix vector.
$\mathbf{H}$	Histogram vector

**Table 4.2.** Workload characterization parameters.

### 4.1.3 Model of Structural Hazards

In the model of structural hazards, we calculate the expected IPC ( $IPC_w$ ) of a workload for a given processor configuration as a function of the global issue window size. The global issue window contains all the instructions that are ready to be dispatched to a functional unit, i.e., that have no data or control dependencies. Thus, only the effects of structural hazards are calculated; data and control dependencies are not considered.

We assume that there are  $w$  ready-to-issue instructions in the global window for our model of structural hazards, where  $0 \leq w \leq NS$ . The state of the instructions is represented by a vector  $\mathbf{M}_w = (m_1, m_2, \dots, m_T)$ , where each element  $m_i$  describes the number of instructions of type  $i$  [serr94,yama94]. The sum of the elements of  $\mathbf{M}_w$  is equal to  $w$ . Given a global issue window of size  $w$  and  $T$  functional types, the total number of states for the window is equal to the number of  $w$ -selections of a  $T$ -set [bose84], or  $\binom{w+T-1}{T-1}$ .

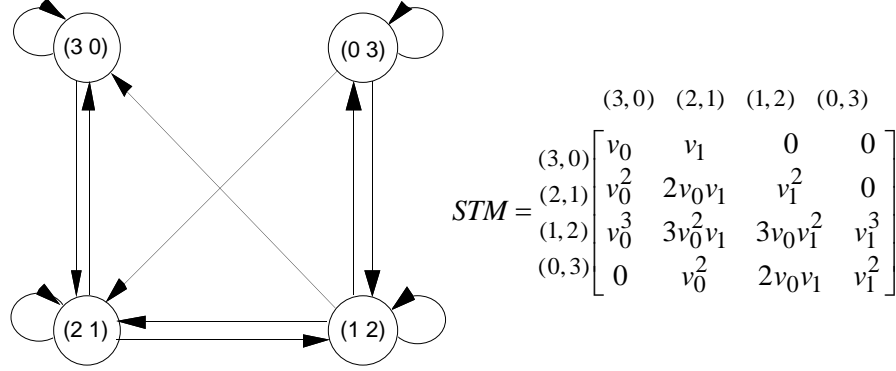
For a global window state  $\mathbf{M}_w$ , the number of instructions dispatched in a given cycle is determined by the number and type of ready-to-issue instructions in the global window and the number and type of functional units. For example, consider a machine with one floating point unit, two integer units ( $\mathbf{C}=(1,2)$ ), and global window size of three ( $w=3$ ). If no floating point and three integer instructions are in the global window ( $\mathbf{M}_3=[0,3]$ ), then only two integer instructions are dispatched in that cycle. For a given functional unit configuration and global window state, the number of instructions of type  $j$  dispatched, referred to as  $i_j$ , is the smaller of the number of functional units of type  $j$  and the number of instructions of type  $j$  within the window. We define  $I_{M_w}$ , the total number of instructions dispatched for the state  $\mathbf{M}_w$  of the window, as the sum of  $i_j$  over all functional unit types:

$$I_{M_w} = \sum_{j=1}^T i_j, \quad i_j = \min(c_j, m_j) \quad (4.2)$$

The expected IPC for a window of  $w$  instructions is the sum of the instructions issued  $I_{M_w}$  times the probability of being in state  $\mathbf{M}_w$ , defined as  $Q_{M_w}$  for all the states of the global window:

$$IPC_w = \sum_{\mathbf{M}_w} I_{M_w} Q_{M_w} \quad (4.3)$$

We obtain  $Q_{M_w}$  by calculating the steady state probabilities of a Markov chain involving the states of the global issue window. The states are not independent due to the relationship between the window size and configuration of functional units. For a given processor configuration and global window state, the next states are determined by the instructions dispatched. For our example, two integer instructions are dispatched and one remains. Therefore, the next state must contain at least one integer instruction ( $[2,1]$ ,  $[1,2]$ , and  $[0,3]$ ). Table 4.3 lists all states, the possible next states, and the number of instructions dispatched



**Figure 4.1.** Markov chain and the state transition matrix (STM) for the example with homogeneous workloads,  $w=3$ ,  $\mathbf{M}_3=[0,3]$ ,  $\mathbf{C}=(1,2)$ .

in each state. Thus, the next state is dependent on the current state of the global window  $\mathbf{M}_w$  and the functional unit configuration  $\mathbf{C}$ .

The state transition probabilities are computed from  $\mathbf{V}$ , the runtime instruction mix for the workload. For our next discussion, we restrict workloads to a single instruction mix even though more than one stream may be executing. Later we discuss workloads of different instruction mixes.

Current State $\mathbf{M}_3$	Possible Next States	$I_{M_3}$
[3, 0]	[3, 0] [2, 1]	1
[2, 1]	[3, 0] [2, 1] [1, 2]	2
[1, 2]	[3, 0] [2, 1] [1, 2] [0, 3]	3
[0, 3]	[2, 1] [1, 2] [0, 3]	2

**Table 4.3.** State table for the example in Figure 4.1.  $I_M$  is the total number of instructions dispatched for the current state.

We assume that instructions of different types are uniformly distributed throughout the execution of the workload. For our example, since two instructions issue in state [0,3], the probability of a transition to state [2,1] is equal to the probability of filling the global issue window with two floating point instructions. This probability, obtained from the runtime instruction mix  $\mathbf{V}=(v_0, v_1)$ , is just the square of the probability  $v_0$ . Figure 4.1 shows the Markov chain model and the state transition matrix  $STM$  for our example.

After solving for the steady state probabilities  $Q_{[x,y]}$ , we calculate the expected IPC for a window of size  $w$ ,  $IPC_w$ , using equation (4.1). For our example, the expected IPC for a window size of three is

$$IPC_3 = I_{[3,0]}Q_{[3,0]} + I_{[2,1]}Q_{[2,1]} + I_{[1,2]}Q_{[1,2]} + I_{[0,3]}Q_{[0,3]}.$$

#### 4.1.4 Model of Data and Control Hazards

The instruction-level-parallelism of a workload is degraded by data and control hazards. An estimated degradation factor obtained from the workload is used to scale the performance obtained by the structural hazard model. This section describes a measure of the instruction-level-parallelism of a workload, i.e., the capacity to issue a number of instructions in a cycle.

The histogram vector  $\mathbf{H} = (h_0, h_1, h_2, \dots, h_S)$  characterizes an active stream issuing part or all of its window of instructions in a cycle. Each element  $h_w$  describes the probability that  $w$  instructions issue in a given cycle. The single stream measure  $\mathbf{H}$  for the benchmark can be obtained with performance monitoring hardware, as in the new POWER2 architecture [welb93]. In our case, we use Powersim to extract the histogram vector.

We define  $P_w$  as the probability of having  $w$  instructions ready-to-issue in a cycle. For the case of two homogeneous workloads (and therefore the same histogram vector), the probability  $P_0$  of zero ready-to-issue instructions is equal to the probability of the two workloads having zero ready-to-issue instructions, or  $h_0^2$ . For example, the following coefficients are obtained with two workloads ( $N=2$ ) and a stream window size of two ( $S=2$ )

$$\begin{aligned} P_0 &= h_0^2, \\ P_1 &= 2h_0h_1, \\ P_2 &= h_1^2 + 2h_0h_2, \\ P_3 &= 2h_1h_2, \\ P_4 &= h_2^2 \end{aligned}$$

For convenience of notation we group all the histogram vectors of the streams in a matrix  $\mathbf{H}$  of dimensions  $N \times S$ . An element  $h_{i,x_i}$  in  $\mathbf{H}$  contains the probability that  $x_i$  instructions from stream  $i$  are issued in a given cycle. In general, for a given number of streams  $N$  and a matrix  $\mathbf{H}$ , the probability that  $w$  instructions are ready to be issued in a cycle is obtained from the probability that each stream  $i$  has  $x_i$  ready-to-issue instructions, such that  $x_1 + x_2 + \dots + x_N = w$ , and  $0 \leq x_i \leq S$ . In other words

$$P_w = \sum_{\substack{x_1+x_2+\dots+x_N=w, \\ 0 \leq x_i \leq S}} h_{1,x_1} h_{2,x_2} \dots h_{N,x_N} \quad (4.4)$$

Estimating the histogram vector is important. Often, the actual values cannot be obtained and must be predicted. For example, the executable code, actual machine configuration, or tools to extract data may not be available. However, since we had the tools to extract this information, we chose to execute the programs and measure the values. A machine configuration with an infinite number of functional units is used to collect the values. The rationale is that the infinite number of functional units will negate any effect of structural hazards, allowing us to measure more accurately the data and control hazards.

#### 4.1.5 Rationale

One of the main problems faced by the analyst is the large state space of many Markov chains, which precludes not only the model solution, but also the generation of the transition rate matrix [souz92]. The storage and time needed for a large non-sparse Markov chain of  $N$  states is  $O(N^2)$  and  $O(N^3)$  respectively.<sup>1</sup> Methods must be devised to reduce the number of states, while retaining the accuracy of the solution.

In our case, a larger Markov chain can be built with all states. The total number of states is equal to the sum of the number of states for all possible  $w$  ready-to-issue instructions, i.e.

$$\sum_{w=0}^{NS} \binom{w+T-1}{T-1} = \binom{NS+T}{T}$$

In contrast, the maximum number of states in our approach occurs when the number of ready-to-issue instructions is equal to the window size, i.e.

$$\binom{NS+T-1}{T-1}$$

Thus, the ratio of the total number of states to the maximum number of states is  $(1 + NS/T)$ . For example, for four streams ( $N=4$ ), four instruction types ( $T=4$ ), and stream window sizes of four ( $S=4$ ), the total number of states is 4844 requiring 188 Mbytes of storage in double precision. Our method requires at most 968 states and 7.7 Mbytes of storage in double precision for a non-sparse state transition matrix.

---

<sup>1</sup>Recent methods have cut down the time complexity to  $O(N^{2.4})$  approximately.

In practice, the number of states can be further reduced by considering only the most significant instruction types. Thus,  $T$  is reduced by selecting the most significant  $v_{unit}/c_{unit}$  values since the less frequent instruction types do not often experience structural conflicts. This is specially true for the bigger window sizes where saturation effects are observed.

Our method applies a decomposition method of large Markov chains [cour77]. The Markov chain is decomposed into smaller chains that are solved independently. The probability of each subchain is computed from the characteristics of the workload. From the way we decompose the chain, we expect the method to be more accurate for streams that don't have enough instruction-level parallelism to saturate the configuration of functional units, or for configurations with many functional units.

## 4.2 Saturation Point

An examination of the graphs in Chapter 3 reveals that the performance gain levels out as the number of streams increases. This is due to the contention for functional unit resources. The phenomenon is most obvious in configuration  $C1$  where saturation occurs when executing as few as two streams. For the larger configurations ( $C2$  and  $C3$ ), saturation occurs when executing a larger number of streams. Performance in a multistreamed processor increases with the number of streams until one or more functional units saturate. The saturation point is an important performance measure indicating the maximum performance for a configuration and a type of schedule.

The saturation point depends on the characteristics of the workloads running on the streams as well as the type of schedule employed for dispatching instructions. We assume an infinite number of streams running in the machine. We review three interesting cases where the saturation point is computed in closed form.

### 4.2.1 Extended Workload Specification Parameters

Our study of saturation considers the characteristics of different thread types, not just homogeneous workloads. We extend the workload specification parameters defined in Section 4.1.2. We define  $K$  as the number of distinct thread types running on the machine. The vector  $\mathbf{W}=(w_1, w_2, \dots, w_k)$  is used to describe the workload of threads of each type. Each

element  $w_{type}$  represents the fraction of each thread  $type$  in the streams of the machine. For example, a machine with four streams loaded with three copies of *hanoi* and one copy of *dhrystone* has the following characterization:  $K=2$ ,  $\mathbf{W}=(w_{hanoi}, w_{dhry})=(0.75, 0.25)$ .

We generalize the instruction vector  $\mathbf{V}$  described in Section 4.1.2 to the instruction mix matrix  $\mathbf{V}$  that specifies the percentage of instructions executed in each functional unit type. Each element  $v_{type,unit}$  of  $\mathbf{V}$  contains the dynamic probability that instructions from a thread  $type$  require execution in a functional  $unit$ . For example, *hanoi*'s instructions are 56% integer, 12% branch, and 32% load/store, e.g.,  $v_{hanoi,integer}=0.56$ ,  $v_{hanoi,branch}=0.12$ ,  $v_{hanoi,memory}=0.32$ . Table 4.4 summarizes the extended workload specification parameters.

$K$	Number of distinct thread types
$\mathbf{W}=(w_1, w_2, \dots, w_k)$	Workload of threads of each type
$\mathbf{V}$	Dynamic instruction mix ( $K \times T$ ) matrix

**Table 4.4.** Extended workload characterization parameters.

## 4.2.2 Homogeneous Workload

We consider the special case of multiple copies of the same thread type loaded on the  $N$  streams of the machine ( $K=1$ ). A simple bottleneck analysis is used for the saturation point. Assume that saturation occurs in some  $unit$  type and that there are  $c_{unit}$  functional units of this type. In saturation the utilization of the units reaches 100 percent; thus, on each cycle  $c_{unit}$  instructions are fed to the  $unit$  type. We define  $IPC_{unit}^{sat}$  as the maximum instructions-per-cycle attainable from a unit type. If  $v_{type,unit}$  is the probability that a thread  $type$  needs a functional  $unit$ , then this performance measure is

$$IPC_{unit}^{sat} = c_{unit} / v_{type,unit}$$

Different unit types have different saturation points. The unit type with the minimum saturation value constitutes the bottleneck of the system. Thus, the overall saturation value is determined by taking the minimum of the saturation values for all functional units. We refer to this result as the saturation value of each thread  $type$ :

$$IPC_{type}^{sat} = \underset{unit=1}{MIN} \frac{c_{unit}}{v_{type,unit}} \quad (4.5)$$

For example, if there are 40% integer instructions in some thread type and one integer unit, the saturation point for the integer unit is  $1/0.40=2.5$ . If there are 60% floating-point instructions and two floating-point units, the saturation point for the floating-point units is  $2/0.6=3.33$ . The integer unit determines saturation, and the overall saturation value is 2.5.

### 4.2.3 Best-Case Schedule

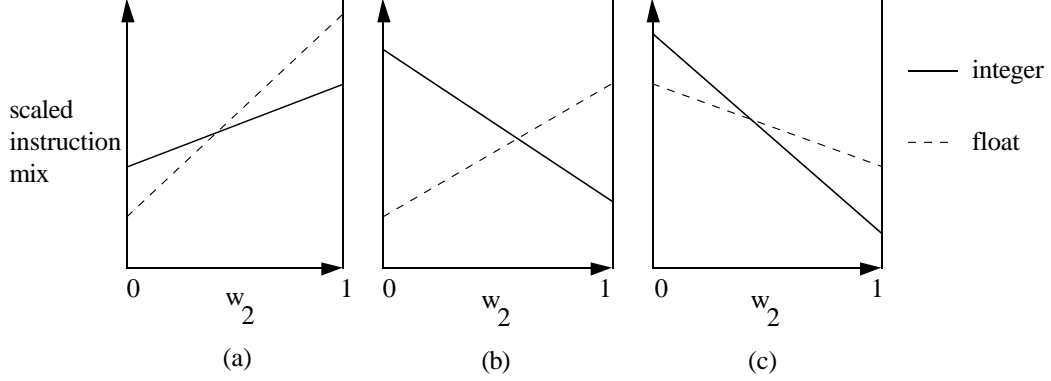
The previous section considered homogeneous workloads. Combining heterogeneous workloads makes better use of the functional units. Higher saturation values could be achieved. For example, consider the case of two heterogeneous thread types: one with only integer instructions, and the other with only floating point instructions. The saturation value of the first thread type is  $c_{int}$  (the number of integer units), while the saturation value of the second thread type is  $c_{float}$  (the number of float units). Combining the two workloads gives a saturation value of  $c_{int}+c_{float}$ , the sum of the individual saturation values.

Here we consider a hypothetical best-case schedule that mixes the benchmark types in such a way as to obtain the highest saturation value. The obtained saturation value is used to compare the optimality of any schedule with the best-case schedule.

Assume that the instruction mixes from the different thread types are mixed in some optimal way, according to some workload  $\mathbf{W}$ . The fraction of instructions sent from a thread *type* to a functional *unit* is equal to the fraction of time of the benchmark type  $w_{type}$ , times the probability  $v_{type,unit}$ . Thus, the combined instruction mix for each *unit* type is equal to the sum over all the thread types, or  $\sum_{type=1}^K w_{type}v_{type,unit}$ . The saturation value for the combined instruction mix is computed using equation (4.5)

$$IPC^{sat} = \underset{unit=1}{MIN} \left( \frac{c_{unit}}{\sum_{type=1}^K w_{type}v_{type,unit}} \right) \quad (4.6)$$

Obtaining the highest value of saturation is as a linear optimization problem. The problem consists in finding the optimal workload  $\mathbf{W}$  such as to maximize the saturation value, or conversely, to minimize the scaled-instruction-mix. The scaled-instruction-mix is defined as the instruction mix divided by the number of functional units of each type. In other words, the problem is



**Figure 4.2.** Example of optimization with two benchmark types.

Find  $\mathbf{W}=(w_1, w_2, \dots, w_k)$  (optimal workload)

such that  $\frac{1}{c_{unit}} \sum_{type=1}^K w_{type} v_{type, unit}, unit=1, \dots, T$  (scaled-instruction-mix) is minimal

subject to:  $0 \leq w_i \leq 1, w_1 + w_2 + \dots + w_K = 1$

For example, if two benchmark types containing only integer and floating-point operations are mixed, the scaled-instruction-mix system of equations reduces to two lines, because  $w_2=1-w_1$ . The optimization problem consists of finding the lowest value for the interval. Figure 4.2 illustrates three possible scenarios of the two lines, as a function of  $w_2$ . For cases (a), and (c), the minimum value is found in the extremes, i.e.  $w_2=0$  and  $w_2=1$ , respectively; thus, one of the benchmarks is excluded. For case (b), the minimum value is found in the intersection of the two lines.

We use the result of this optimization problem to determine the optimality of a schedule. In other words, comparing the saturation values of a determined schedule and the best-case gives the degree of optimality of the schedule.

#### 4.2.4 Fair Schedule

A *fair* schedule attempts to give each stream a proportion of processor time close to  $1/N$  of the total time. In this schedule, the priority is alternated on a cycle-by-cycle basis to each stream in a random fashion. The highest priority stream of a given cycle is allowed to issue as many instructions as possible in that cycle. If free slots in functional units exist, then other streams are allowed to issue instructions to these leftover functional units.

Since the schedule is fair, each thread *type* will have the highest priority for a fraction  $w_{type}$  of the total time on the average. We now assume that all the streams saturate on the same functional *unit*. Each thread type will then execute a number of instructions in the saturation unit proportional to its share  $w_{type}$ , i.e.  $w_{type}c_{unit}$  instructions-per-cycle in the saturation unit. Since the frequency of the instruction type is  $v_{type,unit}$  for each thread type, each thread type will execute  $w_{type}c_{unit}/v_{type,unit}$  instructions-per-cycle. As in Section 4.2.2, the saturation point is determined by taking the minimum over all functional units for the sum of the instructions-per-cycle of all thread types:

$$IPC^{sat} = \underset{unit=1}{MIN} \left( \sum_{type=1}^K \frac{w_{type}c_{unit}}{v_{type,unit}} \right) \quad (4.7)$$

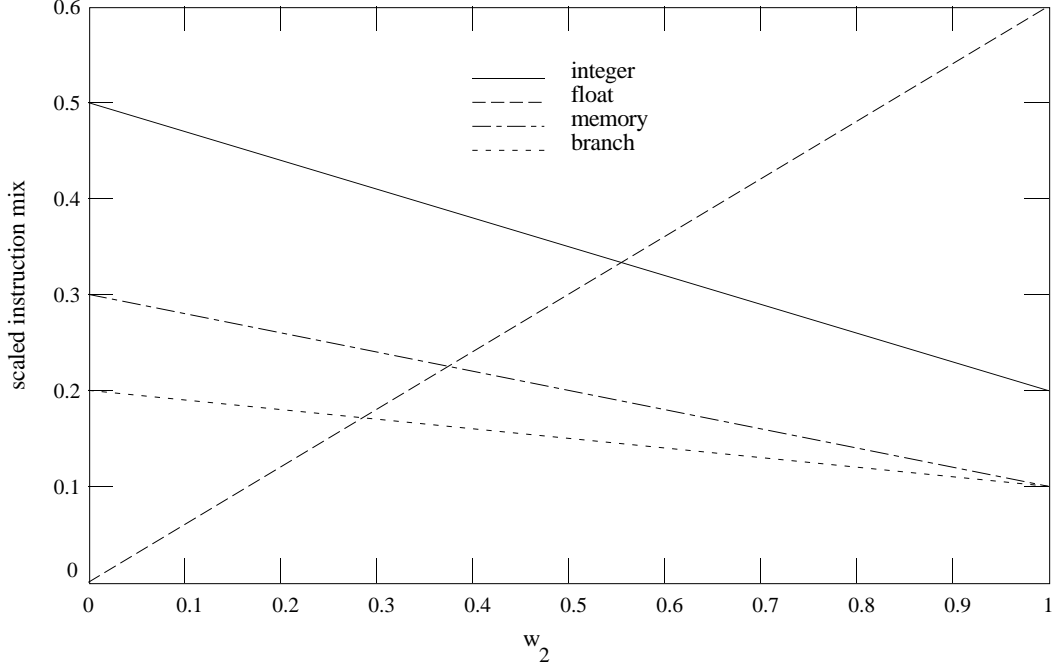
The reader should compare this result to Equation (4.6). Thus, fairness does not necessarily produce the most optimal utilization of the functional units. However, optimality is obtained in some cases, as we will explain later with an example.

It is interesting to see the contribution of each thread type to the total number of instructions issued. Instructions are taken from each stream in the same proportion under low-load conditions, when the number of instructions issued is low compared with the number of functional units. Thus, each thread type issues a fraction  $w_{type}$  of the total number of instructions issued. As saturation is approached, the fraction taken from each thread type changes; we refer to this value as  $w_{type}^{sat}$ . If we define *sat\_unit* as the unit where saturation occurs, then the number of instructions-per-cycle executed by each thread type is  $w_{type}c_{sat\_unit}/v_{type,sat\_unit}$ . The share of each type is computed as the number of instructions-per-cycle executed by the type divided by the total number of instructions:

$$w_{type}^{sat} = \frac{w_{type}/v_{type,sat\_unit}}{\sum_{unit=1} w_{type}/v_{type,unit}}, \quad type = 1, 2, \dots, K \quad (4.8)$$

The rest of the analysis will consider benchmarks that saturate in different functional units. Without loss of generality, we present an example with only two benchmark types.

The analysis is similar to the linear optimization problem described in Section 4.3.2. The scaled-instruction-mix as a function of the workload  $\mathbf{W}$  is drawn as a line for the case of two thread types, since  $w_1 + w_2 = 1$ . Figure 4.3 plots the scaled-instruction-mix as a function of



**Figure 4.3.** Scaled instruction mix for two benchmarks. The figure shows the scaled-instruction-mix (instruction mix divided by the number of functional units) for two thread types, as a function of  $w_2$ , the fraction of thread type 2.

$w_2$ , for the integer, float, memory, and branch instruction types. The value  $w_2=0$  corresponds to the scaled-instruction-mix for thread type 1, while  $w_2=1$  corresponds to the one for thread type 2. As seen from Figure 4.4 the integer instruction mix dominate performance for type 1 ( $w_2=0$ ), while the floating-point instruction mix dominate type 2 ( $w_2=1$ ).

Suppose that in our example we have equal numbers of programs of each thread type, i.e.  $w_1=w_2=0.5$ . We start our search by assuming that saturation occurs in the integer units. We can use Equation (4.4) with  $sat\_unit=integer$  to obtain the share of thread type 2 in the instruction mix in saturation, i.e.  $w_2^{sat}=0.714$ . However, Figure 4.3 shows that for the value 0.714 the floating-point instruction mix determines saturation since its value is higher; therefore, our assumption of saturation in the integer unit is incorrect. In our next step, we assume that saturation occurs in the floating-point units. By using Equation (4.4) again, we obtain  $w_2^{sat}=0$  (by taking the limit when the instruction mix approaches zero). However, this is not correct because the integer instruction mix is higher for such a value. By using an iterative procedure, it can be shown that the correct saturation point converges to the intersection of the integer and float mix lines. Thus, the correct answer is  $w_2^{sat} \approx 0.55$ .

For our example we obtain  $w_2^{sat}$  as a function of  $w_2$  using this procedure:

$$w_2^{sat} = \begin{cases} 10w_2/(4+6w_2) & 0 \leq w_2 \leq 1/3 \\ 5/9 & 1/3 \leq w_2 < 1 \\ 1 & w_2 = 1 \end{cases}$$

$w_2^{sat}$  is a maximum for  $1/3 \leq w_2 \leq 1$ , and is found in the intersection of the integer and float scaled mix lines. This is the saturation point of the best-case schedule.

The procedure solves a linear optimization problem. For the example,  $w_2^{sat}$  is computed. If we determine that this value corresponds to our assumption of saturation for some functional unit, then the result is correct and the search terminates. Otherwise, the search continues until some optimal value is found.

### 4.3 Validation

We validated the analytic technique by comparing its estimations to the results of Powersim. First, we describe our simulation environment and benchmark suite. Next, we present the results from both the analytical technique and the hardware simulator. Finally, we quantify the differences between the model and the simulator and discuss the discrepancies.

We studied the three different hardware configurations shown in Chapter 3, Table 3.5. The scheduler employs a *fair* scheduling algorithm. In this algorithm, priority is alternated among the streams on every cycle in a round-robin fashion. The scheduler dispatches as many instructions from the high-priority stream's window as possible in the current cycle. If *free* slots in functional units still exist, the scheduler attempts to dispatch instructions from the other streams to fill the slots. For our experiments we used a stream issue window of four. This means that the maximum number of instructions that a stream can issue is four, resembling the original RS6000 architecture [groh90].

As explained before, each instruction stream is interpreted as a random stream of instructions of each type. The instruction stream is characterized by its instruction mix, i.e. each instruction type has a probability derived from the instruction mix. Instructions are generated from a binomial distribution using the instruction type probabilities. The accuracy of the model depends on the assumptions that 1) the instruction mix is a stationary process, and 2) instructions appear in the instruction stream in a random manner. We used two estimators ( $\sigma$ ,  $\delta$ ) to measure the degree of closeness to these assumptions:

	hanoi	dhystone	fibonacci	savage	fft	doduc	li
skew $\delta$	0.134	0.143	0.165	0.199	0.132	0.314	0.198
integer	56.1 $\pm$ 0.1	46.5 $\pm$ 0.0	48.4 $\pm$ 0.0	15.1 $\pm$ 0.2	23.2 $\pm$ 0.9	17.3 $\pm$ 4.0	36.4 $\pm$ 6.5
float	0	0	0	38.1 $\pm$ 0.2	16.8 $\pm$ 3.5	27.9 $\pm$ 4.8	0
cr logic	0	0.9	0	2.5 $\pm$ 0.0	0	3.3 $\pm$ 1.4	1.3 $\pm$ 0.0
branch	12.2 $\pm$ 0.0	20.0 $\pm$ 0.0	22.6 $\pm$ 0.0	9.2 $\pm$ 0.1	18.9 $\pm$ 1.5	10.2 $\pm$ 3.1	21.2 $\pm$ 1.2
memory	31.7 $\pm$ 0.1	31.5 $\pm$ 0.1	29.0 $\pm$ 0.0	35.1 $\pm$ 0.1	58.1 $\pm$ 1.3	41.2 $\pm$ 3.9	41.0 $\pm$ 6.9

**Table 4.5.** Workload characteristics. The skew and the percentage (mean  $\pm$  standard deviation) of each instruction type. Values were sampled at fixed intervals of 1600 cycles.

- $\sigma$  The standard deviation of each instruction type computed from samples collected periodically from the simulations.
- $\delta$  The skew measure of the degree of randomness of the workload

$$\delta = \sum_{i=1}^T \sum_{j=1}^T v_i |v_j - t_{i,j}|$$

where  $v_i$  is the probability of the instruction type  $i$ , and  $t_{i,j}$  is the probability of transition from instruction type  $i$  to instruction type  $j$ , measured by the simulator. The skew measure is zero for a perfectly random workload because  $v_j$  must be equal to  $t_{i,j}$ . The skew measure approaches one for a deterministic workload. Thus, the skew is a good indicator of the fraction of randomness of a workload.

### 4.3.1 Benchmarks

A set of common benchmark programs is the workload. We selected the small benchmarks *hanoi*, *dhystone*, *fibonacci*, *savage*, and *fft*. From the SPEC'89 benchmark suit we selected *doduc* and *li*. All the benchmarks ran to completion, with the exception of *doduc* and *li* which ran with traces of 150 million of instructions each. Table 4.5 shows the characteristics of the workloads evaluated. The numbers indicate the mean and standard deviation of each instruction type (mean  $\pm$   $\sigma$ ) in percentage, and the skew of each benchmark ( $\delta$ ). Integer multiply and divide instruction mixes are not shown since they constitute a very small percentage of the instructions.

### 4.3.2 Comparison to Simulation Results

The first experiment consisted in running homogeneous workloads, with multiple copies of the same thread loaded in all streams of the machines. For example, we ran *hanoi* with one, two, three, and four copies of *hanoi* in the streams of the machine.

The number of instructions-per-cycle issued (IPC) are shown in Table 4.6 for one, two, three, and four streams. Table 4.6 compares the predicted and simulation results for homogeneous workloads and shows the percentage of error of the prediction. As seen from the results, the benchmarks with the smallest error are *hanoi* and *dhrystone*. They correspond to the benchmarks with the smallest skew and standard deviation values. In any case, the peak error does not exceed eight percent for all the configurations and benchmarks. The average error is smaller for configuration C3 where the number of resources is bigger than what an individual stream may use.

The previous results used homogeneous workloads. Combining heterogeneous workloads makes better use of the functional units. Higher saturation values could be achieved. We expect that the model gives results closer to the simulations, since the behavior of a heterogeneous workload is expected to be more random than a homogeneous workload.

Although the model was explained with homogeneous workloads, the model can be extended to model heterogeneous workloads. This is accomplished by using an averaging technique over the heterogeneous workload to obtain average homogeneous workload characteristics ( $V_{average}$ ), using the instruction mix at saturation. These characteristics are entered into the analytical model to obtain performance estimates of heterogeneous workloads. Table 4.7 shows the results for heterogeneous workloads. We selected pairs of benchmarks and ran simulations for one, two, three, and four pairs, (2, 4, 6, 8 streams). For example, for the experiment with four streams, two copies of *hanoi* and *dhrystone* were loaded in the four streams of the machine.

### 4.3.3 Discussion

Our results show that the analytical technique produces estimates very close to those of the simulation results. Over the 168 total estimates, the average deviation from the simulation results was 2.3%. On average, the analytic technique produced fairly even results across the various configurations and number of streams employed within a workload.

benchmark	streams	C1			C2			C3		
		predicted	simul.	error %	predicted	simul.	error %	predicted	simul.	error %
hanoi	1	1.55	1.52	1.94	2.34	2.37	-0.93	2.56	2.56	-0.10
	2	1.76	1.76	-0.17	3.33	3.31	0.44	5.02	5.13	-2.05
	3	1.78	1.78	-0.04	3.53	3.53	-0.12	7.05	6.96	1.38
	4	1.78	1.78	0.00	3.56	3.56	-0.03	8.28	8.14	1.77
dhystone	1	1.59	1.64	-3.29	2.15	2.21	-2.58	2.34	2.36	-0.89
	2	2.01	2.03	-0.98	3.35	3.41	-1.73	4.54	4.55	-0.40
	3	2.12	2.12	-0.05	3.89	3.90	-0.36	6.35	6.39	-0.63
	4	2.14	2.14	0.05	4.12	4.12	0.00	7.62	7.67	-0.59
fibonacci	1	1.86	2.02	-7.75	2.58	2.72	-5.18	2.96	2.99	-0.73
	2	2.05	2.07	-0.97	3.62	3.86	-6.26	5.61	5.70	-1.58
	3	2.07	2.07	-0.07	3.91	4.08	-4.16	7.42	7.61	-2.45
	4	2.07	2.07	0.00	4.02	4.12	-2.61	8.30	8.63	-3.79
savage	1	1.43	1.51	-5.53	1.78	1.81	-1.42	1.83	1.83	-0.15
	2	2.07	2.16	-4.43	3.18	3.30	-3.57	3.57	3.60	-0.66
	3	2.34	2.38	-1.69	4.07	4.21	-3.15	5.04	5.16	-2.41
	4	2.46	2.48	-0.74	4.59	4.65	-1.43	6.13	6.29	-2.61
fft	1	1.16	1.22	-5.16	1.48	1.44	2.70	1.52	1.51	0.26
	2	1.57	1.66	-5.23	2.55	2.61	-1.95	2.93	2.94	-0.26
	3	1.68	1.72	-2.07	3.11	3.23	-3.85	4.00	4.09	-2.26
	4	1.71	1.73	-0.82	3.33	3.42	-2.41	4.65	4.81	-3.48
doduc	1	1.56	1.57	-0.64	2.04	2.01	1.49	2.12	2.17	-2.30
	2	2.15	2.11	1.90	3.50	3.44	1.74	4.08	4.05	0.74
	3	2.34	2.30	1.74	4.29	4.16	3.13	5.59	5.53	1.08
	4	2.41	2.36	2.12	4.65	4.50	3.33	6.54	6.41	2.03
li	1	1.77	1.68	5.36	2.42	2.32	4.31	2.73	2.67	2.25
	2	2.22	2.06	7.77	3.65	3.44	6.10	5.03	4.88	3.07
	3	2.35	2.19	7.31	4.16	3.90	6.67	6.40	6.19	3.39
	4	2.40	2.26	6.19	4.41	4.26	3.52	7.01	6.79	3.24

**Table 4.6.** Results for homogeneous workloads. Predicted IPC and the simulation IPC for each workload, configurations C1, C2, and C3, and for one, two, three, and four streams.

Table 4.8 shows the average percentage of error for different configurations and workloads for homogeneous workloads. While the model makes many generalizations about the architecture and workload that may not seem to be representative of most programs, our results show that the predicted performance is close to the simulation results for all configurations in six of the seven benchmarks (*hanoi*, *dhystone*, *fibonacci*, *savage*, *fft*, *doduc*). Also, the error is lower for configuration C3 than for configuration C1, confirming that the error diminishes as the configuration grows in number of functional units.

benchmarks	streams	C2			C3		
		predicted	simulation	error %	predicted	simulation	error %
hanoi - dhrystone.	2	3.41	3.40	0.17	4.79	4.81	-0.50
	4	3.90	3.87	0.90	8.07	8.04	0.41
	6	3.93	3.91	0.69	9.19	9.09	1.12
	8	3.94	3.92	0.49	9.44	9.37	0.69
hanoi - fft	2	3.31	3.30	0.34	3.99	3.94	1.17
	4	4.24	4.25	-0.24	6.60	6.63	-0.46
	6	4.45	4.50	-1.11	7.24	7.29	-0.65
	8	4.54	4.59	-1.21	7.31	7.35	-0.63
dhrystone - fft	2	3.19	3.27	-2.25	3.77	3.78	-0.14
	4	4.37	4.40	-0.80	6.35	6.49	-2.17
	6	4.69	4.65	0.91	7.19	7.22	-0.39
	8	4.80	4.73	1.49	7.33	7.29	0.49
fibonacci - savage	2	3.90	3.99	-2.13	4.70	4.75	-1.09
	4	5.27	5.29	-0.39	7.99	8.18	-2.33
	6	5.68	5.76	-1.44	9.13	9.28	-1.59
	8	5.85	5.99	-2.39	9.38	9.44	-0.62
fibonacci - fft	2	3.55	3.77	-5.68	4.37	4.43	-1.28
	4	4.55	4.80	-5.15	7.19	7.40	-2.81
	6	4.80	4.95	-3.04	7.70	7.78	-1.11
	8	4.90	4.99	-1.81	7.74	7.76	-0.29

**Table 4.7.** Results for heterogeneous workloads. Predicted IPC and simulation IPC for combinations of two benchmarks. Results shown for two, four, six, and eight streams taking two of each benchmark at a time. Configurations C2 and C3 were evaluated.

benchmark	C1	C2	C3	AVERAGE
hanoi	0.54	0.38	1.33	0.75
dhry	1.09	1.17	0.63	0.96
fibonacci	2.20	4.55	2.14	2.96
savage	3.10	2.39	1.46	2.32
fft	3.32	2.73	1.57	2.54
doduc	1.60	2.42	1.54	1.85
li	6.66	5.15	2.99	4.93
AVERAGE	2.64	2.68	1.66	

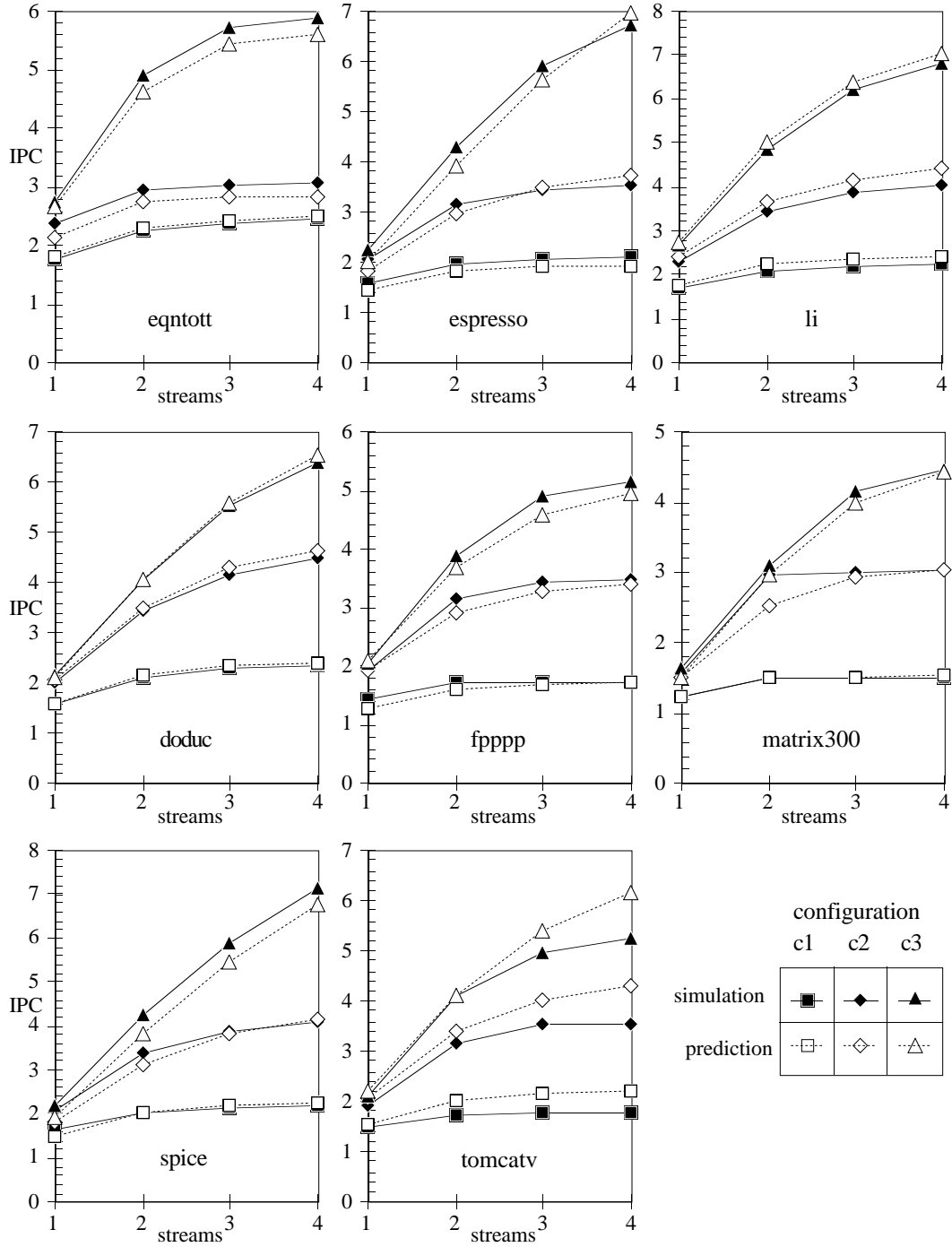
**Table 4.8.** Error for homogeneous workloads. The table shows the average percentage error for each benchmark and configurations C1, C2, and C3.

Table 4.9 summarizes the errors for heterogeneous workloads. When benchmarks are combined the instruction mix behaves more randomly, thus the errors we obtain are smaller. For example, the combination of *hanoi* and *fft* produces less error than either benchmark considered individually. We also see that the performance of heterogeneous workloads in some cases is better than the individual benchmarks. For example, when we run four streams in configuration C2, the combination of *hanoi-fft* has an IPC of 4.25 compared to 3.56 and 3.42 for *hanoi* and *fft* respectively.

benchmarks	C2	C3	AVERAGE
<i>hanoi - dhystone</i>	0.56	0.68	0.62
<i>hanoi - fft</i>	0.72	0.73	0.73
<i>dhystone - fft</i>	1.36	0.80	1.08
<i>fibonacci - savage</i>	1.59	1.41	1.50
<i>fibonacci - fft</i>	3.92	1.37	2.65
AVERAGE	1.63	1.00	

**Table 4.9.** Error for heterogeneous workloads. The table shows the average percentage of error for combinations of two benchmarks and configurations C2 and C3.

We tried our method with most of the SPEC'89 benchmarks to assess the differences between prediction and simulation. Figure 4.4 compares the simulation results to the prediction results for the SPEC benchmarks running homogeneous workloads. In spite of the assumptions of our analytic model, the prediction shows good correlation with the simulation results, with a few exceptions. For example, *matrix300* shows linear growth with the number of streams until the performance saturates. In fact, *matrix300* spends more than 95% of the time in a small loop of only 5 instructions. We observed that instructions from different streams coming from small loops tend to rearrange so as to produce a linear growth with the number of streams. Thus, the behavior of *matrix300* is more deterministic than random. We have observed this behavior in a few small loops that have few instructions, for example, *double*, *short*, and *int* from the *Byte* benchmark set.



**Figure 4.4.** Comparing simulation and prediction with homogeneous workloads for the SPEC benchmarks.

	eqntott	espresso	li	doduc	fpppp	matrix300	spice	tomcatv
skew $\delta$	0.557	0.247	0.198	0.314	0.284	0.464	0.202	0.183
integer	41.0 $\pm$ 4.2	50.9 $\pm$ 4.4	36.4 $\pm$ 6.5	17.3 $\pm$ 4.0	3.3 $\pm$ 4.8	1.4 $\pm$ 0.1	42.1 $\pm$ 6.9	5.2 $\pm$ 3.0
float	0	0	0	27.9 $\pm$ 4.8	35.6 $\pm$ 4.4	16.3 $\pm$ 0.5	2.5 $\pm$ 5.4	44.3 $\pm$ 16.
cr logic	0	3.3 $\pm$ 1.4	0	3.3 $\pm$ 1.4	0	0	0	1.7 $\pm$ 4.7
branch	36.3 $\pm$ 2.4	10.2 $\pm$ 3.1	25.4 $\pm$ 6.4	10.2 $\pm$ 3.1	2.1 $\pm$ 2.5	16.7 $\pm$ 0.4	20.1 $\pm$ 5.8	6.8 $\pm$ 7.4
memory	22.7 $\pm$ 5.1	41.2 $\pm$ 3.8	23.3 $\pm$ 6.1	41.2 $\pm$ 3.8	58.6 $\pm$ 0.3	65.6 $\pm$ 3.8	34.2 $\pm$ 5.1	42.0 $\pm$ 18.

**Table 4.10.** SPEC'89 characteristics. Skew, and percentage (mean  $\pm$  std. deviation) of each instruction type. Values were sampled at fixed intervals of 1600 cycles.

In most cases, the prediction is a lower performance bound because of the conservative assumptions of the model. In some cases, for example *tomcatv*, the prediction is higher than the actual performance. The discrepancy in *tomcatv* is caused by the program spending some time in a loop followed by another loop. Thus, the two loops have different behavior; an average technique cannot predict properly the performance. The high standard deviation of *tomcatv* reflects this behavior, as seen in Table 4.10. The best way to predict performance in *tomcatv* is to divide the time spent by the benchmark in two sections, such that prediction can be performed independently for the two sections; the instruction mix for each of the sections is more stationary. This technique can be applied to benchmarks such as *whetstone*, which runs consecutive timing loops, since the prediction using an average mix is quite inaccurate.

## 5 A Multistreamed Instruction Issue Mechanism

The issue mechanism guarantees that instruction results are maintained according to a sequential model of computation. Thus, the mechanism deals with data, control, and structural hazards for an adequate timing to correctly issue instructions to the functional units. We present a novel instruction issue mechanism for a distributed multistreamed superscalar processor that addresses the following aspects:

- Components that can be shared by the streams or private to each stream.
- How data, control, and structural conflicts are resolved.
- The distribution of functions in the issue mechanism.
- Minimization of global interconnections, relaxing of synchronization constraints, decoupling of instructions and data, and the use of the data locality principle, as explained in sections 2.3.1 and 2.3.2.
- Minimization of global communication by using information compression.
- Implementation of precise interrupts.
- Implementation of speculative execution with instruction squashing.
- The variations in the mechanism as a result of tradeoffs in the design.

Since the imprecise interrupt problem could be caused by dynamic scheduling, it is desirable to treat dynamic scheduling and implementation of precise interrupts together. Also, the distributed nature of the processor needs a distributed approach for instruction issue and implementation of precise interrupts. We present an algorithm that is a variant of Tomasulo's algorithm [toma67] and the Dispatch Stack [torn84]. Our scheme eliminates the need for a reservation station and storage for tags associated with each register. However, our scheme is limited to few functional units and streams because of the rising complexity of the interconnection network.

Section 5.1 reviews the necessary background. Section 5.2 discusses the components of the mechanism. Section 5.3 discusses the rules to issue instructions given many types of conflicts. Finally, Section 5.4 briefly discusses the implementation of precise interrupts at a coarser-level.

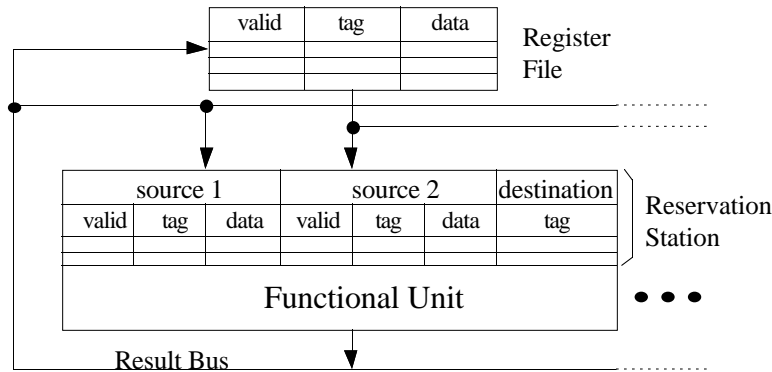


Figure 5.1. Hardware implementation for Tomasulo's algorithm.

## 5.1 Background

We briefly discuss the most relevant literature. For more information, the reader may consult [dwy91,john91,kato92,henn90,hwu87,park91,smit88,sohi90,toma67,torn94,uht92].

### Data Conflicts

Data dependency conflicts may occur when storage locations are reused. Working registers, memory locations, and special registers are examples of storage locations. For example 1) an instruction may depend on a previous instruction if the second instruction deposits data into a location read by the first instruction, 2) two instructions may update the same storage location. Conflicts of this sort need to be properly resolved.

Data conflicts occur when there are non-null intersections in the domain and range of instructions. The domain  $X$  of an instruction is the set of objects containing data needed by the instruction, and the range  $Y$  of an instruction is the set of objects modified by the instruction. Let  $S=\{1,2,\dots,N\}$  be the set of instructions in the dynamic instruction stream ordered by position in the dynamic instruction stream, where instruction  $i$  occurs before instruction  $j$  if  $i < j$ . Three data conflicts can occur: a) RAW (read-after-write) if  $X_j \cap Y_i \neq \emptyset$ , b) WAR (write-after-read) if  $Y_j \cap X_i \neq \emptyset$ , and c) WAW (write-after-write) if  $Y_j \cap Y_i \neq \emptyset$ .

### Tomasulo's Algorithm

Tomasulo's algorithm [toma67] uses data forwarding and register renaming techniques to resolve data conflicts. Figure 5.1 shows a block diagram of the hardware interconnection. Each register has (*valid*, *tag*, *data*) fields. The *tag* is a copy of the tag of the latest instruction that will modify the register. When an instruction issues, an identification tag is

assigned from the tag pool, the new tag replaces the *tag* field of the destination register of the issuing instruction, and the *valid* bit is cleared. The assigned tag returns to the tag pool for future reuse when the instruction completes execution.

Each functional unit has a reservation station (RS). An instruction is dispatched to the appropriate functional unit if there is an empty entry in its RS. The (*valid, tag, data*) fields of the instruction's source registers are sent to the corresponding entry of the RS. The instruction can execute if all the source operands are *valid*; otherwise, it will wait until this happens. When an instruction finishes execution in a functional unit, the result *data* and *tag* are broadcast to all RSs and the register file. If there is a match with any tag, the result data is written into the data field of the matched entry, and the valid bit of the corresponding entry is set. As a result, some instructions waiting in RSs become ready to execute.

WAW and WAR hazards are eliminated because all instructions that read a determined register issue before a subsequent instruction that writes the same register. When an instruction writes to a register, all previously issued instructions that read the same register have copies of either the data or the reservation station's name that will produce the data. RAW hazards are also eliminated by the sequential nature of instruction tag assignment.

### Dispatch Stack

The dispatch stack (DS) [torn84] is a centralized scheme using an instruction window for data dependency resolution. In the DS, dependency resolution occurs *before* issue, unlike Tomasulo's *after* issue. Data dependencies are checked with two usage counters ( $\alpha, \beta$ ) for each register, representing the number of times that the register is designated as a destination ( $\alpha$ ) and source ( $\beta$ ) register for the preceding but unfinished instructions. Initially, the counters are set to zero. Each entry in the instruction window has several fields:

ITag	OP	S <sub>1</sub>	$\alpha$	S <sub>2</sub>	$\alpha$	D	$\alpha$	$\beta$
------	----	----------------	----------	----------------	----------	---	----------	---------

The  $S_1$ ,  $S_2$ , and  $D$  fields represent the source and destination registers. The  $\alpha(S_1)$  and  $\alpha(S_2)$  counter fields indicate the RAW data conflict, the  $\alpha(D)$  counter field indicates the WAW data conflict, and the  $\beta(D)$  counter field indicates the WAR data conflict. These fields are not simple storage, but counters. An instruction can issue from the window only if all of its four counters are equal to zero.

When a new decoded instruction enters the instruction issue window, the  $\alpha$  and  $\beta$  counter fields for the new entry are set with the current counter values of the corresponding

registers. The  $\alpha$  and  $\beta$  counters of the corresponding registers are then incremented. These incremented counter values are used for subsequent instructions. When an instruction completes, register usage counters and the corresponding counter fields in the instruction issue window are decremented.

A difference with Tomasulo's algorithm is the method of passing data from functional units to waiting instructions. The DS passes data through the register file; Tomasulo's algorithm forwards data to an instruction holding buffer (the RS).

### **Precise Interrupts**

Interrupt execution is precise if the processor state can be reconstructed to the point of the interrupt assuming a sequential computation model. This means that all the preceding instructions previous to the interrupting instruction executed and updated the process state correctly but the interrupting instruction itself and its subsequent instructions did not affect the process state. Since an imprecise interrupt can leave the machine in an unrecoverable state, an interrupt needs to be precise.

If all the instructions are forced to modify the process state in program order, the interrupts are precise. However, this limits performance. For high-performance processors where instructions can complete out-of-order, a hardware mechanism must be provided to keep precise interrupts.

### **Reorder Buffer**

The reorder buffer [pope91,smit88] is a queue to store results produced by instructions that complete out-of-order. When an instruction is decoded, an entry is reserved for the instruction and the index number of the reserved entry is attached to the instruction as a tag. When an instruction completes, its result and error status are written to the corresponding entry of the reorder buffer. An instruction updates the architectural register file with its result after all preceding instructions finished and updated the register file, thereby in-order system updates are ensured. The entry at the head of the reorder buffer is checked every cycle. If its result is ready and there is no interrupt, the result updates the register file. However, if the instruction is interrupted, the result as well as the result of all subsequent instructions are discarded and the interrupt is handled.

The number of cycles from the time the instruction issues to the time its result updates the register file can be large because of the in-order update. To avoid halting instructions waiting for source data, results in the reorder buffer can be bypassed to functional units before they are written to the register file.

## 5.2 Components of the Issue Mechanism

### 5.2.1 Pipeline Structure

Figure 2.4 explains the hardware organization to support several instruction streams running concurrently. Instructions are brought by the fetch unit from the instruction cache and stored in each of the stream prefetch buffers. Instructions are sent to the window where the logic determines data dependencies. The scheduler dispatches instructions free of dependencies to the appropriate functional unit provided structural hazards do not exist. Instructions are dispatched to the functional units through an issue interconnection network (ICN).

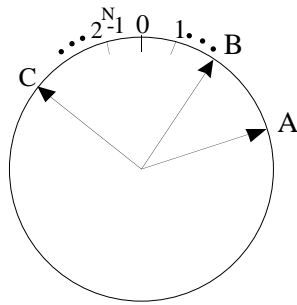
A modified RISC pipeline structure to process instructions in a multistreamed processor is based on the following processing stages:

- IF** Instruction fetch and predecode. The instruction is brought from the instruction cache and deposited into instruction buffers. Branches are detected. Predecoding is performed for data dependency resolution in the next stage.
- SC** Instruction scheduling. Instructions dependencies are determined and resolved in the instruction window.
- ICN** Instructions are forwarded to the appropriate functional units using the appropriate interconnection network.
- ID** Instruction decode. Instructions are decoded in the functional unit; operands are fetched from the data storage (for example the register file).
- EX** Instructions are executed.
- WB** Results are written back to the data storage.

### 5.2.2 Instruction Tag Unit

To maintain precise interrupts, a way to identify the sequential order of appearance of instructions in the instruction stream must be used. Our algorithm assigns consecutive tags to instructions. The tags are obtained from an instruction counter that is updated as new instructions appear. Using this unique tag, the sequential order can be reconstructed regardless of the place where instructions and results are.

We use modulo arithmetic for tag assignment. Figure 5.2 shows that the relative modulo- $N$  order of instructions A, B, C is  $A > B > C$ , i.e. A occurs after B and B occurs after C in the



**Figure 5.2.** Example of tags in the instruction window.

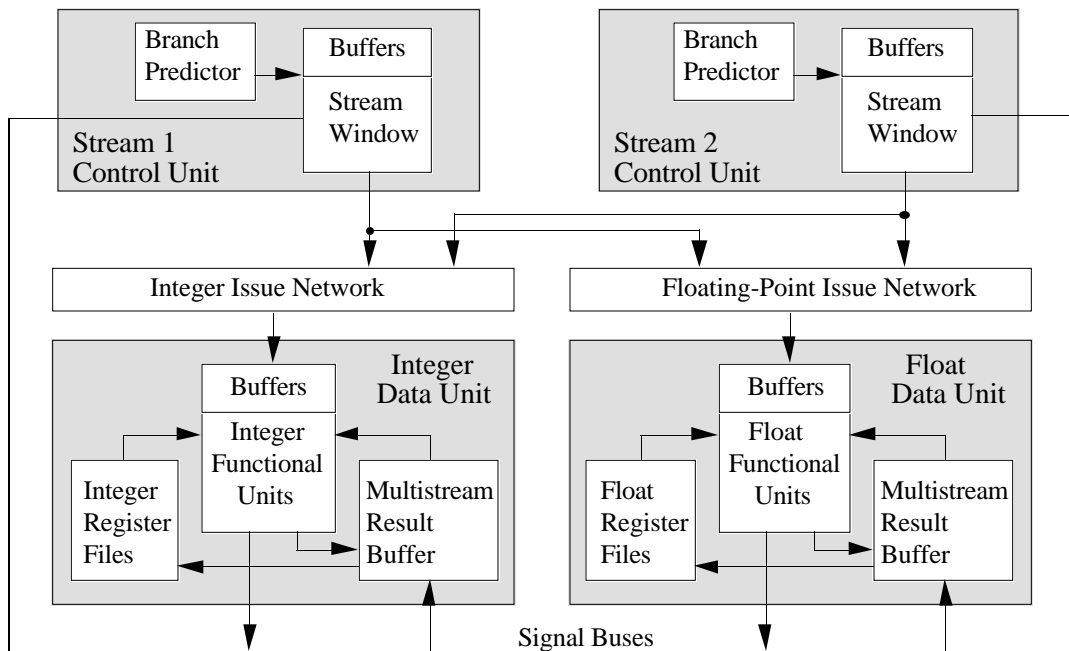
instruction stream. In several situations, it is not only necessary to identify if the instruction tag matches any stored tag, but also to identify the relative order of two tags, namely an instruction is classified as *before*, *equal*, or *after* a reference instruction. This is an extension to the match logic where the results of interest are *equal*, or *not equal*.

The module- $N$  comparator is an ordinary arithmetic subtractor, provided that the maximum tag distance is less than half of the window size, i.e.  $|Max - Min| < 2^{N-1}$ , where  $Max$  and  $Min$  are the maximum and minimum instruction tags. If this limit is exceeded instruction decoding will stop until old tags are released. A tag counter of four bits is enough in most cases because it is unlikely that more than seven instructions are active at a time. Note that in many situations it suffices to determine whether an instruction is *before* or *equal/after* another instruction (or *before/equal* or *after*), so only the most significant bit of the subtractor is implemented. Thus, the comparator can be implemented with majority gates [mukh86], which can be embedded in the cell design.

The comparator logic is a key factor to reduce tag traffic, since communication is one of the most important bottlenecks in superscalar processors. For example, a single tag broadcast can invalidate a group of instructions whose tags are *equal/after* some reference tag.

### 5.2.3 General Block Diagram

Figure 5.3 shows an example of the multistreamed hardware for two streams and two data types (integer and floating-point); instruction cache, data cache, and load/store units are not shown. The block diagram contains four components: a) the control units where instructions are stored and scheduled, b) the issue network used to forward instructions from the control units to the data units, c) the data units where instructions are executed, and d) the signal buses between the control and data units.



**Figure 5.3.** Example of multistreamed hardware.

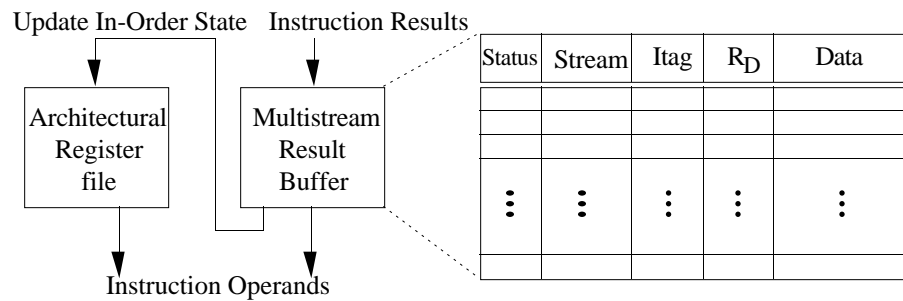
Each control unit has instruction buffers, an instruction window, branch predictor hardware, and an instruction tag unit (not shown in the figure). Each control unit

- Fetch instructions from the instruction cache and store them in instruction buffers.
- Predict branches and modify the instruction fetch target according to the prediction.
- Invalidate instructions and results of instructions that follow a mispredicted branch.
- Detect data dependencies (explained in Section 5.2.7).
- Forward instructions to the appropriate data units.
- Keep track and maintain the precise state of the instruction stream.

Each data unit has one or more functional units, a multistreamed architectural register file, and one or more result buffers. The data units

- Decode instructions, fetch operands, and execute instructions.
- Forward operands to instructions waiting for operands, using pipeline bypasses.
- Generate broadcast invalidation signals resulting from internal exceptions.
- Commit results to the architectural register file on command from the control unit.

The issue networks forward instructions from the control units to the data units. A scheduling policy is used to issue instructions from all the streams, detect structural hazards, and allocate functional units to instructions.



**Figure 5.4.** The multistreamed result buffer.

Signal buses connect the data units to the control units. The buses

- Return instruction tags to the control units once instructions complete execution.
- Broadcast commit signals to move results from the result buffers to the register files.
- Broadcast invalidation signals resulting from interrupts or mispredicted branches.

### 5.2.4 The Result Buffer

Result buffers maintain precise interrupts by holding intermediate results produced by the functional units. Then, the intermediate results are committed to the respective architectural register files which hold the precise state.

The scheduler must make sure that there is a free entry in the result buffer before an instruction can be issued. The free entry is assigned to the issued instruction and marked *waiting*. Once the instruction result is ready, the result is stored in the entry and the entry marked *valid*. The result buffer releases the entry and updates the architectural register file upon request from the control unit. Figure 5.4 shows the fields of each entry.

- *Status*: Bits to indicate that the entry is valid, waiting, or free.
- *ITag*: tag of the instruction. This field uses comparative logic (see Section 5.1.6).
- *$R_D$* : destination register. This field uses match logic.
- *Data*: instruction results.

The result buffer is a multiported design with many simultaneous accesses in a cycle. Fortunately, the accesses to the reorder buffer can be prioritized such that ports can be used for several functions. The priorities are (1 is the highest, 4 the lowest):

- 1) Store results. This is a high priority operation to avoid freezing functional unit pipelines.

2) Read operands. Some operands needed by an instruction may be in the result buffer. To bypass data, it may be necessary to check whether the operands of a decoded instruction are available in the result buffer.

3) Update the precise state. The control unit sends a message requesting the architectural register file to be updated from the reorder buffer with all the instructions whose tag is before the *Itag*. This operation can be delayed. However, it may increase the need to read operands from the result buffer.

4) Invalidate entries. Instruction results are invalidated when exceptions (internal and external) are produced or because of a mispredicted branch.

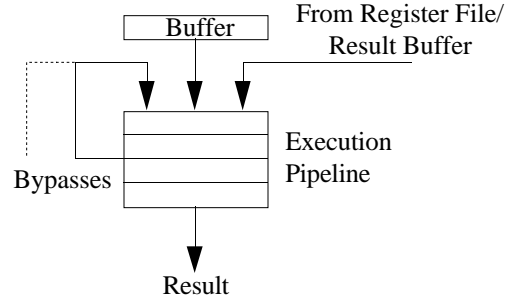
Sharing the result buffer among the streams has the advantage that while the number of entries produced by each stream vary with the cycle, the total number of intermediate results is approximately constant with the number of streams, i.e. it depends more on the number of functional units that produce results. This means that the utilization is more uniform than the utilization of separate stream reorder buffers.

On the other hand, sharing increases the pressure from simultaneous accesses to the result buffer. The number of ports (read,write) of the multistreamed result buffer depends on the number of functional units and the number of streams accessing it. State-of-the-art technology implements ten-port register files [hara92]. Given current technology limiting the number of ports, it is unlikely than more than a few functional units are connected to the same result buffer. Also, the higher the number of functional units, the higher the complexity of the data network needed to feed operands to the functional units. A possible solution to reduce the port pressure is to adopt a hierarchical connection scheme. For example, a reduced number of streams share a few functional units; several units remain private to each stream. Also, a reduced number of functional units share a result buffer.

The result buffer is based on the reorder buffer [smit88]; however, the result buffer does not need to be a queue, because of the ordering effect of the instruction tags.

The result buffer can resolve most of the data conflicts between instructions:

- WAW hazards are resolved because results are ordered by the instruction tag. Note that updating the architectural register file requires conflict resolution logic because two or more instructions can be writing to the same register. In this case, the most recent result updates the register file; other results are discarded.
- WAR hazards are resolved since results are ordered by the instruction tag.



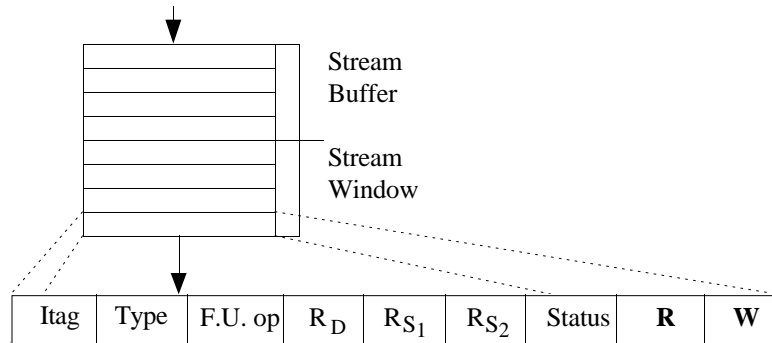
**Figure 5.5.** Block diagram of functional units.

- RAW hazards are resolved in a limited form. An instruction may need to perform an associative search for an operand in the result buffer. There could be results for that register from instructions before or after the current instruction. Instructions always need data produced by the most recent “before” instruction. Thus, we limit the number of “before” instructions to one, since the comparing logic can only tell if an instruction is *before* or *after* another, not the relative distance between them.

### 5.2.5 Functional Units

Figure 5.5 shows a general block diagram for a functional unit in the multistreamed processor. We assume pipelined functional units organized in groups that process the same data type. Each functional unit may use buffers for decoupling purposes. The buffers may be used to decode instructions, or for more sophisticated functions such as reservation stations. Functional units read operands from the architectural register file, the reorder buffer, or (optionally) from bypasses connected to functional units. However, bypasses are expensive to implement because 1) an array of  $N$  functional units needs  $N^2$  bypasses, and 2) determining if a bypass contains the needed operand may require tag matching logic.

Functional unit pipelines do not have freeze logic in order not to compromise the response time of individual streams in a multistreamed environment. DEC's Alpha [digi92] is an architecture without freeze logic in the execution pipelines; this choice simplifies the timing of the pipeline stages. Thus, each pipeline produces results at a rate of one per cycle. A result coming out from the pipeline is discarded if there is no *waiting* entry for the corresponding instruction tag, *Itag*, in the result buffer. This may happen if a previous



**Figure 5.6.** The stream window.

invalidation signal marks the entry *free* in the result buffer. A buffer may be needed to store temporarily the result if the write ports of the result buffer are busy.

If any error is produced, the functional unit broadcasts an invalidate signal to all data sections and the stream control section, indicating that all the instructions "equal or after" the faulting instruction are canceled. As far as the data units are concerned, this is the only action needed. The control section decides at a later stage when to take the interrupt.

### 5.2.6 Stream Window

The stream window is the centralized issue control. It decides about instruction issue and maintains the precise state of the stream. The stream window accepts instructions from the buffers. Branches are folded, i.e. they are removed from the instruction buffers so they do not enter the window. The stream window is a queue that holds instructions in different phases: stalled for dependencies, ready-to-issue, issued, and executed. Executed instructions are retired from the front of the queue to maintain the precise stream state.

Figure 5.6 shows the fields of each entry. *Status* is the instruction status, *Itag* is the instruction tag, *Type* is the instruction type, *F.U. op* is the functional unit op code,  $(R_D, R_{S1}, R_{S2})$  are the destination and source registers, and  $(\mathbf{R}, \mathbf{W})$  are the read and write vectors (explained in next section). The number of entries in the window depends on:

- The instruction fetch bandwidth, i.e. the maximum, average, and minimum number of instructions that can be simultaneously fetched from the instruction cache.
- The number of functional units that can execute these instructions.
- The complexity of the register conflict circuit used for data hazard detection.
- Whether branch prediction/speculation is used across several branch levels.

- The parallelism of the applications.

The complexity of the register conflict circuit grows rapidly with the increase in the window size. Define  $N_{RAW}(w)$ ,  $N_{WAW}(w)$ , and  $N_{WAR}(w)$  as the number of register RAW, WAW, and WAR hazards among several instructions for a window of size  $w$ . The total number of hazards is  $N_{DH} = N_{RAW} + N_{WAW} + N_{WAR}$ . The size of the register conflict sets for a typical RISC instruction with two source registers and one destination register is

$$N_{RAW}(w) = w(w-1), \quad N_{WAW}(w) = w(w-1), \quad N_{WAR}(w) = w(w-1)/2$$

Therefore, the size of the register conflict set could grow to  $N_{DH} = 5w(w-1)/2$ . As  $w$  increases the hardware circuit complexity gets worse and/or the cycle time gets larger (tradeoffs can be made); therefore, for simplicity of the design small windows sizes are typically chosen.

### 5.2.7 Determining the Register Conflict Set

Two instructions are register independent if no register dependencies exists between them. A set of instructions is register independent if they are pairwise register independent. Register independence is a necessary (but not sufficient) condition for the out-of-order issue of instructions. Other dependencies related to hardware resources and storage operations need to be considered.

We formalize the method developed by Dwyer [dwy91] for determining the register conflict set. The conflict set is determined in two steps in the **IF** and **SC** pipeline stages. Two binary vectors  $R_i$  and  $W_i$  are computed in the IF stage for each fetched instruction  $i$ :

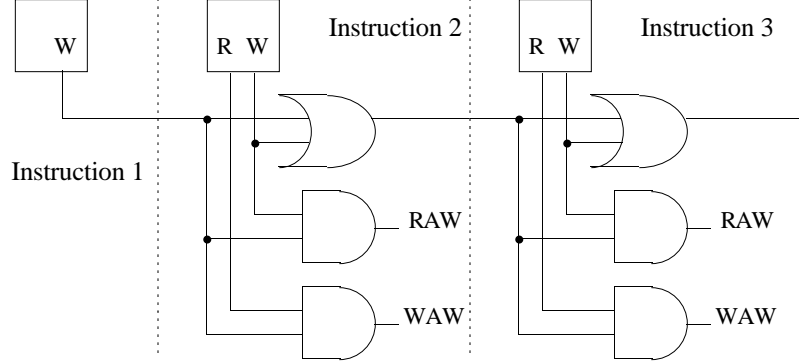
$$\begin{aligned} R_i &= \bigvee_{u \in U_i} \text{decode}(u, S) \\ W_i &= \bigvee_{v \in V_i} \text{decode}(v, S) \end{aligned} \quad (5.1)$$

Where  $U_i$  and  $V_i$  are the read and write register sets,  $\bigvee$  is the bitwise-or operator, and *decode* transforms a register binary code in a binary vector of length  $S$ , where  $S$  is the size of the register file. For example, if an instruction reads registers {4,6} and writes register 2, and there are 8 registers, then (bit 0 is the MSB)

$$U_i = \{4,6\}, \quad V_i = \{2\}, \quad R_i = (0,0,0,0,1,0,1,0,0), \quad W_i = (0,0,1,0,0,0,0,0,0).$$

The register hazard vectors are computed in the **SC** stage with the help of two associative semigroup computations:

$$W_{i,j} = \bigvee_{k=i}^j W_k, \quad R_{i,j} = \bigvee_{k=i}^j R_k, \quad i \leq j \quad (5.2)$$



**Figure 5.7.** Computing the WAW and RAW register conflict set for a small window.

The register hazard vectors are determined as:

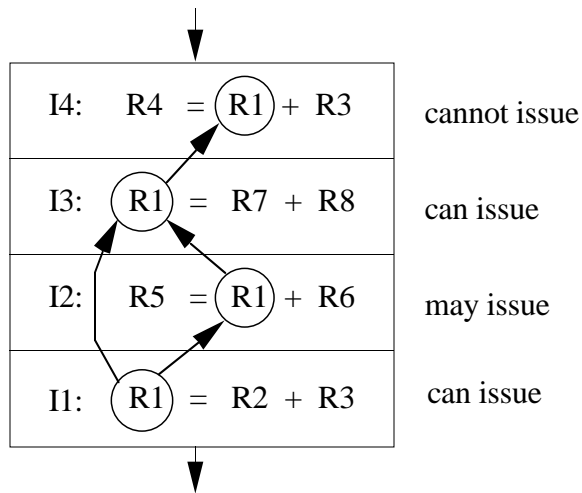
$$\begin{aligned}
 WAW_{i,j} &= \begin{cases} \mathbf{0}, & i \geq j \\ W_j \wedge W_{i,\dots,j-1}, & i < j \end{cases} \\
 RAW_{i,j} &= \begin{cases} \mathbf{0}, & i \geq j \\ R_j \wedge W_{i,\dots,j-1}, & i < j \end{cases} \\
 WAR_{i,j} &= \begin{cases} \mathbf{0}, & i \geq j \\ W_j \wedge R_{i,\dots,j-1}, & i < j \end{cases}
 \end{aligned} \tag{5.3}$$

An instruction  $j$  has a register WAW hazard if  $WAW_{i,j} \neq \mathbf{0}$  where  $i$  is the first instruction in the window ( $i < j$ ). Similarly, an instruction has a RAW hazard if  $RAW_{i,j} \neq \mathbf{0}$ , and a WAR hazard if  $WAR_{i,j} \neq \mathbf{0}$ . Note that there are two basic semigroup computations, since the RAW semigroup vector can be computed from the  $W_{i,j}$  semigroup vector. We will show in Section 5.3 that only the  $W$  semigroup vector is needed. Figure 5.7 shows a single or-propagate scheme to compute the RAW/WAW vectors that can be used for small window sizes.

## 5.3 Issue Mechanism

### 5.3.1 Issue Rules

We already mentioned the data dependency resolution properties of the result buffer. This makes it unnecessary to solve many of the dependencies at the stream window. In particular, there are only two situations when an instruction cannot or may not issue:



**Figure 5.8.** Example of dependency graph for a window of four instructions.

- An instruction cannot issue if there are a RAW hazard at the instruction and a WAW hazard at the previous instruction, i.e.  $(RAW_{i,j} \neq 0) \wedge (WAW_{i,j-1} \neq 0)$ , because at most one result from a previous instruction can be in the result buffer (Section 5.1.5).
- An instruction may or may not issue if there is a RAW hazard at the instruction but there is not a WAW hazard at the previous instruction, i.e.  $(RAW_{i,j} \neq 0) \wedge (WAW_{i,j-1} = 0)$ . This depends on the available hardware resources. If the functional units have reservation stations (RS), then the instruction may issue because the instruction may wait in the RS until its operands are available. Even without RS, an instruction may issue if the conflict involves only one register, i.e. if  $((\neq / (RAW_{i,j} = 1) \wedge (WAW_{i,j-1} = 0))$ , where ' $\neq$ ' adds the elements of the vector. The instruction may issue because the operand from the register in conflict may be obtained from a functional unit bypass mechanism or from the result buffer.

Figure 5.8 shows an example of the issue rules for register dependencies. Instruction *I1* is the oldest instruction in the window. *I3* can issue in spite of the WAW and WAR hazards, because the result buffers resolve these dependencies. *I2* may issue but it may need to wait until *R1* from *I1* is computed. Finally, *I4* cannot issue because the result buffer holds two entries for *R1* and there is not way to determine that the needed data comes from the most recent instruction *I3*.

### 5.3.2 Instruction Issue Format

The instruction issue format is kept to a minimum to simplify the issue networks. The instruction format uses a maximum of 40 bits for a POWER instruction and two streams:

Stream	Itag	F.U. Op	R <sub>D</sub>	H <sub>S1</sub>	R <sub>S1</sub>	H <sub>S2</sub>	R <sub>S2</sub> /Immediate
--------	------	---------	----------------	-----------------	-----------------	-----------------	----------------------------

- *Stream, Itag* identify an instruction from a stream (5 bits)
- *F.U. Op*: Functional unit operation code (6 bits).
- *R<sub>D</sub>, R<sub>S1</sub>, R<sub>S2</sub>/immediate*: destination and source registers. The second source operand could be a 16-bit short immediate (5+5+16=26 bits).
- *H<sub>S1</sub>, H<sub>S2</sub>*: optional hint bits (one or two) indicating the location of a source operand.

The hint bits are optional because operands can be found by simultaneous access of the register file and the result buffer. If the operand is not in the result buffer, then the operand is in the register file. The hint bits improve the efficiency of locating operands:

- 1) An operand  $k$  for instruction  $j$  is in the architectural register file if  $RAW_{i,j}^{(k)}=0$ , where  $A^{(k)}$  is the  $k$ -element of vector  $A$ . This is true because the instruction that produced the operand has completed, left the window, and updated the register file.
- 2) An operand  $k$  for instruction  $j$  may be in the result buffer or in a pipeline bypass if  $RAW_{i,j}^{(k)} \neq 0$ .
- 3) An instruction  $j$  with a RAW hazard on an instruction  $i$  may issue one or two cycles after  $i$ , if the plan is to use a pipeline bypass. However, this requires additional counters in the window to keep track of the number of cycles elapsed since the instruction issued.

### 5.3.3 Register Renaming

Register renaming is automatic in Tomasulo's algorithm since RAW, WAW, and WAR hazards are resolved in the reservation stations. Our issue rules limit register renaming. In particular, instructions which overwrite operands can complete out-of-order, but instructions that need the operands must wait in the presence of a conflict.

Better register renaming can be provided by adding a renaming field to the field specifying a register. Thus  $(D,R)$  specifies a register in the result buffer, where  $D$  is the register field and  $R$  is the renaming field; however, only  $D$  updates the architectural register file. For example, assume a one-bit renaming field ( $|R|=1$ ) and that the following four instructions in the window have a conflict in register  $R2$  (Itag=1 is the oldest instruction)

ITag	Write	Read
4		R2
3	R2	
2		R2
1	R2	

Instructions 1 and 3 write *R2*, and instructions 2 and 4 read *R2*. *R2* is identified in instructions 2 and 3 with a '0' rename field. Since there is a WAW conflict at instruction 3, the rename field for register *R2* is changed to '1' for all subsequent instructions. Therefore, instruction 4 can issue in spite of its RAW hazard and the WAW hazard at instruction 3. However, subsequent instructions with conflicts cannot issue since the rename field length is 1-bit. All the subsequent references to *R2* will use '1' for the rename field, until another conflicts changes the rename field back to '0'. Thus, the renaming field works as a counter.

#### 5.3.4 Control Hazards

Branch instructions are usually folded or removed from the instruction buffers before they enter the instruction window. Instructions following a conditional branch whose outcomes are not determined can conditionally issue to the functional units. All that is needed is to record the *Itag* of the first conditionally issued instruction. If the branch prediction turns out to be wrong, an invalidate signal for all the instructions equal/after the *Itag* is broadcast. This scheme can be extended to an arbitrary number of speculated levels by keeping track of the *Itag* at each of the branch boundaries.

#### 5.3.5 Memory Hazards

Performance can be improved by executing concurrently several memory instructions. However, two or more load/store instructions executing out-of-order may have data memory address conflicts. Since these conflicts cannot be determined in the window until the instructions execute, a conservative approach would change the scheduling pattern to in-order issue by considering the instruction type patterns. Many methods such as run-time disambiguation have been proposed. However, these techniques may not be too effective at the object code level, because some new instructions may have to be inserted in the object code to detect alias resolution.

We can relax this conservative scheme if we assume that instructions are not self-modified, thus only data dependency resolution is needed:

- *Load* instructions can issue and complete out-of-order if there are not *store* instructions in the window.
- *Store* instructions can issue out-of-order but need to complete in-order (write to memory) to maintain the precise state of the stream.
- A *load* instruction can issue only after all the preceding *store* instructions have issued. After issue, the *load* must wait until the effective addresses of the preceding *store* instructions are computed. Then, the *load* must check its effective address against the effective address of the *store* instructions. If there is any conflict, the *load* stalls until all the *store* instructions with address conflict finish execution. In particular, the hardware is simpler if a *load* issues only if there is at most one preceding *store* in the window and the preceding *store* has stored its effective address somewhere for the *load* to check for a memory conflict. The reason is that there could be more than one address conflict between the *load* and two or more previous *store* instructions. With our tag comparing scheme, it is impossible to determine if the *load* needs the data that correspond to the most recent *store* to the same location.

The memory or cache must be updated in program order to keep precise interrupts. This is the function of the *store buffer*, with similar functions to a result buffer. The fields in each entry of the store buffer are

status	Itag	address	data
--------	------	---------	------

*Itag* is the tag of the store instruction (comparative logic). *Address* is the effective address (associative logic). *Status* indicates that the entry is valid, invalid, or reserved by a *store* that has not finished computing its effective address. Subsequent *stores* do not conflict, because they are ordered by the *Itag*. Subsequent *loads* compare their effective address looking for a match with a previous *store*. If there is a match, the *load* is delayed by inserting it in a waiting queue. The *load* can execute only after the data from the conflicting *stores* are sent to memory.

### 5.3.6 Exception Rules

There can be several exceptions produced by each stream as instructions execute. The exception mechanism must recognize only the earliest exception and discard the remaining exceptions. The exception mechanism resides in the control unit, where the sequential order is recorded. Internal interrupts are handled in this order:

1. If an instruction generates an error status, the functional unit processing the instruction broadcasts an invalidation signal containing the instruction *Itag*.
2. The invalidation signal reaches the data units, where the results for the instructions that have an instruction tag equal or after *Itag* are annulled.
3. The invalidation signal reaches the stream window, where no action is taken if there is at least one unfinished instruction preceding the faulting instruction. Instead, the error status is stored in the status field of the instruction with the same *Itag*. The control unit may free the locations of all subsequent instructions and start fetching the interrupt service routine.
4. The stream window takes action when the faulting instruction reaches the top of the window, i.e. it is the oldest instruction in the window. The program counter (PC) of the instruction is saved in a register and the PC takes the address of the interrupt service vector. Thus, only the interrupt at the earliest instruction is serviced.

The reader may note that in step 4 the stream window could start dispatching out-of-order instructions of the interrupt service routine before the exception reaches the top of the window. However, the problem is that memory instructions from the interrupt routine may use TLB entries that have not been updated. Thus, step 4 guarantees that the processor is synchronized before the interrupt routine starts (the SYNCH instruction does the same).

External asynchronous interrupts are handled in a similar way to internal exceptions, the only difference is that the oldest instruction in the window is associated with the interrupt so fast interrupt service is guaranteed. Branch squashing is handled in this order:

1. If the branch prediction turns out to be incorrect, the branch predictor broadcasts an invalidate signal containing the branch *Itag*.
2. The invalidation signal reaches the data units, where all the results for the instructions with tag equal or after *Itag* are invalidated.
3. The invalidation signal reaches the stream window, where all the instructions with tag equal or after *Itag* are immediately purged from the window.

Instruction type	Frequency instruction	Exception	Frequency exception	Needs to be precise ?
•most integer instructions •unconditional branches •resolved conditional branches	high	no	-	-
speculated branches	low	yes	low	yes
integer division	very low	yes	low	language dependent
floating point	medium	optional	low	language dependent
load/store	medium	yes	high	yes

**Table 5.1.** Classification of instructions by interrupts.

## 5.4 Relaxing Precise Interrupts

So far, our discussion has concentrated in the implementation of precise interrupts at the finest-granularity, i.e. at instruction level. The implementation of precise interrupts can often be relaxed. Many instructions do not produce interrupts, or asynchronous interrupts such as Reset do not need precise state since they are catastrophic events. Table 5.1 shows a classification of instructions according to interrupts. For example, floating-point interrupts may not be precise because many languages, Fortran for example, do not provide a mechanism for exception recovery for floating-point exceptions. Processors such as Power [groh90] allow the disabling of floating-point exceptions. The compiler must insert code to check exceptions at intervals and execute repair code if needed. For example, the AIX Fortran compiler inserts a trap 'barrier' instruction before an integer division. The trap tests whether the divisor is equal to zero and traps to the operating system if this is the case. In another example, the compiler can also insert special instructions to inform the hardware of maintaining the precise state at some points. Typically, only load/store instructions produce interrupts that need to be handled in hardware completely. Schemes can be devised to take advantage of this fact.

The granularity of the interrupt implementation can be reduced. One idea is to consider load/store and branches as *interrupt barriers*. We can identify a block of instructions with a single tag to reduce the issue granularity. Table 5.2 shows an example. Block 2 starts with a memory instruction. Block 3 starts with the instruction following a conditional branch (BC) so while the branch itself is completed, the instructions following the branch may need to be

reexecuted if the branch is not properly predicted. Thus, a load/store is associated with a barrier for its own block, while a branch is associated with a barrier for the next block.

Instructions associated with the first barrier, and before a second barrier, can issue out-of-order. Assume the instructions of the first barrier issue with the corresponding block tag. These instructions are canceled if there is an exception. The idea is to raise the barrier as early as possible; for example, it can be determined if a load/store will produce a page fault early in the address translation process. If the barrier is raised, the instructions in the block associated with the barrier can write results directly to the architectural register file. There is no need to use a result buffer. However, the issue is restricted because RAW, WAW, and WAR hazards must be determined in the instruction window before an instruction can issue, because of the lower tag granularity. This scheme still requires that the software inserts code to check for floating-point exceptions and take corrective actions if needed.●

group	tag	instruction
1	1	ADD R1,R2,R3
	2	SUB R4,R5,R6
2	3	LOAD R1,R2
	4	CMP R1,R2
	5	MUL R3,R1,R2
	6	ADD R3,R8,R9
	7	BC GE,label
3	8	ADD R1,R2,R3
	9	SUB R4,R5,R6

**Table 5.2.** Example of tagging blocks of instructions.

## 6 Performance Tradeoffs of Multistreamed Architectures

Integration of a group of processors on a single chip provides opportunities for better utilization of resources. A multistreamed processor is a group of processors in proximity that time-share hardware resources such as the cache, instruction decoder, functional units, the interconnection network, and the bus interface.

Hardware sharing causes extra overhead for scheduling, communication, and resource conflict resolution. Also, the interconnect network poses a significant problem as the number of shared resources increases. In some cases hardware sharing is advantageous. For example, cache sharing simplifies maintaining data coherence in a group of caches. However, the contention for the shared resources can cause performance degradation.

These examples illustrate that the optimal configuration of the multistreamed hardware depends on many factors such as the workload characteristics, communication costs, and implementation technology, to name a few.

We examine several of the problems of sharing different hardware modules. We assume a small group of shared resources, such that the interconnection complexity is not significant. We propose solutions to reduce the extra overhead cost, and to improve the performance degradation caused by sharing. The aspects we consider are instruction scheduling and cache sharing.

We present two studies of tradeoffs in a multistreamed superscalar processor. Section 6.1 studies the problem of instruction scheduling and proposes a novel arbitration scheme to reduce the critical path delay. Finally, Section 6.2 studies the problems of sharing the cache and studies the performance of victim caching holding multithreaded contexts.

### 6.1 Scheduling Instructions

The scheduler is the hardware that dispatches instructions from the control section to the data section. Scheduling instructions in a multistreamed architecture introduces overhead and complexity to the dispatching mechanism, because 1) the streams need to use arbitration hardware before they can use the functional units, and 2) an instruction dispatching network

is needed. While a detailed cost-performance tradeoff analysis can be done to determine the best configuration, we concentrate in ways of reducing the overhead of multistreaming (for example, reduction of the critical path delay) while still providing enough instructions to the functional units to maintain them busy.

The objectives of an efficient dynamic scheduler for a multistreamed architecture are summarized in two phrases: 1) provide an efficient issue mechanism when the processor is running with a single thread, and 2) maintain an affordable instruction scheduling scheme when multiple threads are running. This section concentrates on providing an efficient instruction arbitration mechanism. Our study does not consider the effect of the increasing complexity of the instruction interconnection network; therefore, we are limited to use a few functional units of each type.

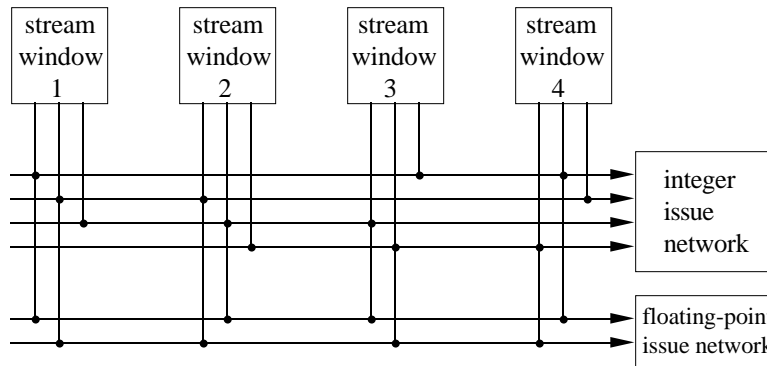
### 6.1.1 Instruction Network

Each of the data types needs a specialized instruction network. The network for every data type transports data among functional units and register files of the same data type. The instruction network sends several instructions per cycle to the functional units to maintain a fast instruction issue rate. Thus, the higher the number of functional units, the higher the bandwidth of the instruction network needs to be. Two kinds of networks are commonly used: bus and multistage interconnection network. We assume a multistreamed computer with four or fewer functional units such that a bus structure can meet the performance requirements.

Figure 6.1 shows an example of an interconnection network with four integer units and two floating-point units. The objective of the scheduler is to send a *long integer* instruction word composed of four integer instructions to the integer units, and two floating-point instructions to the floating point units. In the figure, the maximum number of instructions that each stream can issue is three, to limit the number of buses leaving each stream window.

The scheduling network uses a port architecture. The dispatcher transfers instructions and tags from the stream window to ports that are gateways to the interconnection network. There are four integer ports and two floating-point ports in Figure 6.1. Section 5.3.1 described the format used to carry instruction information (about 40 bits per instruction). Thus, a bus of 160 bits feeds the integer functional units with instructions.

Each stream window arbitrates the use of a port for each ready-to-issue instruction. Arbitration for the port is overlapped with instruction transfer to increase performance. Each



**Figure 6.1.** The scheduling network.

port has an associated instruction type. A port may be free or busy, depending on whether the corresponding functional unit is busy.

Instructions are sent to free ports. Instructions are assigned ports based on precedence. If the number of instructions to issue outnumber free ports of the right types, some instructions are temporarily denied issue.

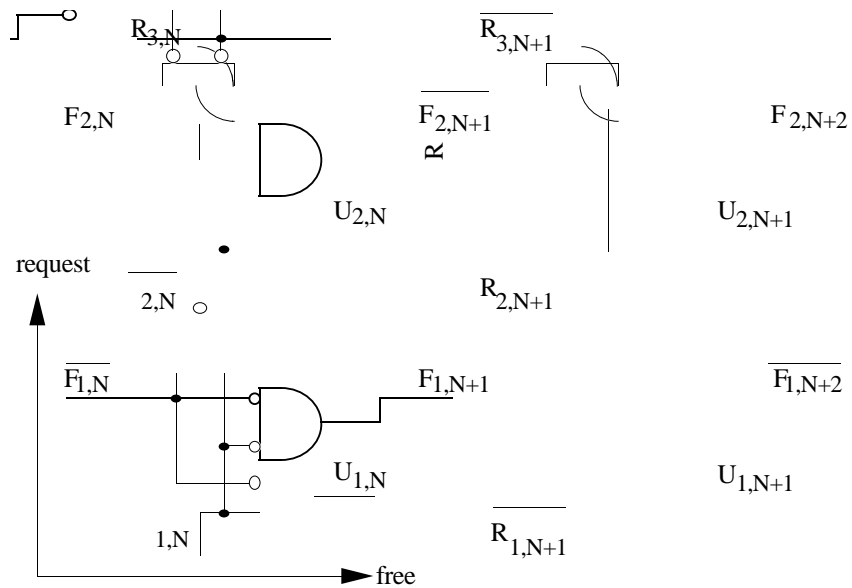
### 6.1.2 Single Stream Arbitration

We present a solution for arbitration of ports with a single stream. We assume that ports are assigned using a precedence scheme. Two steps are needed for an instruction to use a port: 1) identify the instructions that may use ports of an instruction type, and 2) arbitrate the use of these ports using a priority scheme.

In our discussion we assume ports of a single type. The matrix  $\mathbf{R}$  characterizes the requests for ports. The status of ports of some type (free or busy) are characterized by the free matrix  $\mathbf{F}$ . Arbitration produces the use matrix  $\mathbf{U}$ , which indicates the assignments from instructions to ports.

Formally, the request matrix  $\mathbf{R}=\{R_{i,n}\}$  indicates that instruction  $n$  is requesting port  $i$ .  $\mathbf{F}=\{F_{i,n}\}$  indicates the status of the port  $i$  (free or busy) before instruction  $n$  is considered, and  $\mathbf{U}=\{U_{i,n}\}$  indicates the assignment of instruction  $n$  to port  $i$ .

Instructions and ports have assignment precedence. The instruction at the top (oldest) of the window has the highest precedence. We also assume that ports have assignment precedence. The instruction of highest precedence issues through port  $i$ , where  $i$  is the lowest port number that is free. The instruction of second highest precedence issues through port  $j$ ,



**Figure 6.2.** Wavefront computation cell for port arbitration.

where  $j$  is the lowest free port higher than  $i$ , and so on. Inhibit lines are cascaded through the arbitration logic to prevent simultaneous instruction issue through the same port.

The recurrence equations for arbitration are

$$\begin{aligned}
 F_{i,n+1} &= F_{i,n} \overline{R_{i,n}} \\
 R_{i+1,n} &= \overline{F_{i,n}} R_{i,n} \\
 U_{i,n} &= F_{i,n} R_{i,n}
 \end{aligned} \tag{6.1}$$

The first equation indicates that port  $i$  will be free at instruction  $n+1$  if the port is free at instruction  $n$ , and instruction  $n$  does not request the port. The second equation indicates that the request of instruction  $n$  for a port will be propagated to port  $i+1$  if port  $i$  is not free. Finally, the third equation indicates that instruction  $n$  will acquire port  $i$ , if the instruction requested the port, and the port is free.

Figure 6.2 shows an implementation of the recurrence equations (6.1) using standard CMOS NAND/NOR gates. Note the alternating pattern of active low and high signals for the requests and port status. This is to take advantage of a single gate delay per element. Since the *width* of the window (number of instructions) is generally larger than the *height* (number of ports), the circuit is optimized in the *free* direction. Thus, the propagation gates in the *free* direction are larger than the other two gates. Figure 6.2 shows the critical delay

path in thick lines. Each of the large gates in the critical delay path drives three gates: a large one and two small ones (with less than half of the input capacitance of the large gate). Thus, the fan-out of the large gate is less than two.

If the requests are computed with the help of the data dependency analysis network shown in Figure 5.7, arbitration using the circuit of Figure 6.2 can be performed in parallel. Computation is performed in a two-dimensional wave front manner, in the request and free directions. Thus, the critical delay for both data dependency analysis and arbitration is  $W+U+C$  gate delays, where  $W$  is the window size,  $U$  is the number of functional units of a particular type, and  $C$  is a small constant expressing the delay to determine the request from the data dependency analysis. For example, the critical delay for a window of size 8 ( $W=8$ ) and two functional units ( $U=2$ ) is 12 gate delays:

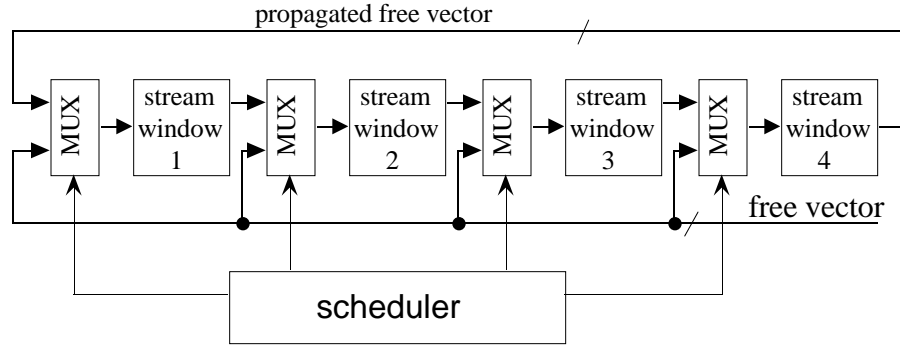
- 8 gate delays for the RAW, and WAW vectors of the last instruction in the window.
- $C=2$  gate delays to generate a request from the RAW, WAW.
- 2 gate delays to propagate port assignments.

### 6.1.3 Multistream Arbitration

Multistreaming requires a larger arbitration network for all the stream windows. Thus, the critical delay path is increased compared to single stream arbitration.

Figure 6.3 shows the interconnection of the stream windows for arbitration of the ports. The scheduler distributes the free vector  $\mathbf{F}$  to the stream windows. Each stream window uses an input multiplexer to take the stream's  $\mathbf{F}$  vector from the previous stream window's output  $\mathbf{F}$ , or directly from the port's  $\mathbf{F}$  vector. For example, the free vector  $\mathbf{F}$  is fed to stream window 1 in a cycle, thus stream 1 has the higher priority in using the ports. Stream window 2 takes the  $\mathbf{F}$  vector from stream window 1; thus, it has the second highest priority. The  $\mathbf{F}$  vector is propagated until it reaches the last stream window. Thus, the last window has the lowest priority.

The scheduler selects in each cycle the stream with the highest priority. The stream with the highest priority will be allowed to arbitrate for all the ports serving the stream. The remaining streams will be allowed to use the ports left free, following the priority chain. The next section presents a technique for reducing the critical delay path for this scheme.



**Figure 6.3.** Port arbitration in a multistreamed processor.

### 6.1.4 Reducing the Critical Path

Unfortunately, if we employ the propagation network proposed in Figure 6.2, the critical path for the multistream arbitration of Figure 6.3 is several times the one for a single stream arbitration. The delay is 36 gate delays for a typical example with windows of size eight, two functional units, and four streams. To reduce the delay of the critical path we could

- Limit the number of stream windows for port arbitration in each cycle. For example, two streams could be selected for arbitration in a cycle, chosen among the streams with the highest number of ready-to-issue instructions.
- Arrange for each port to be used by a subset of all the streams. For example, in Figure 6.1 each integer port is used by at most three stream windows, thus reducing the arbitration critical delay to three stream windows.
- Adopt a look-ahead scheme to reduce the critical path using redundant circuit.

### 6.1.5. A Look-Ahead Scheme for Arbitration

We discuss a look-ahead scheme for computation of the free vector. We want a solution that separates the request matrix  $\mathbf{R}$  from the free matrix  $\mathbf{F}$ , such that the requests for all the streams can be processed in parallel as much as possible.

It can be shown that a solution of the recurrence equations (6.1) for the  $\mathbf{F}$  matrix is

$$F_{i,m+1} = F_{i,n}(C_{n,m}^{(0)} + C_{n,m}^{(1)}G_{n,i-1}^{(1)} + \dots + C_{n,m}^{(i-1)}G_{n,i-1}^{(i-1)}), \quad n < m \quad (6.2)$$

where  $C_{n,m}^{(i)}$  means that are most  $i$  instructions are ready-to-issue in the window of instructions  $(n,m)$  and  $G_{n,k}^{(i)}$  means that at least  $i$  ports are free before instruction  $n$ , in the set

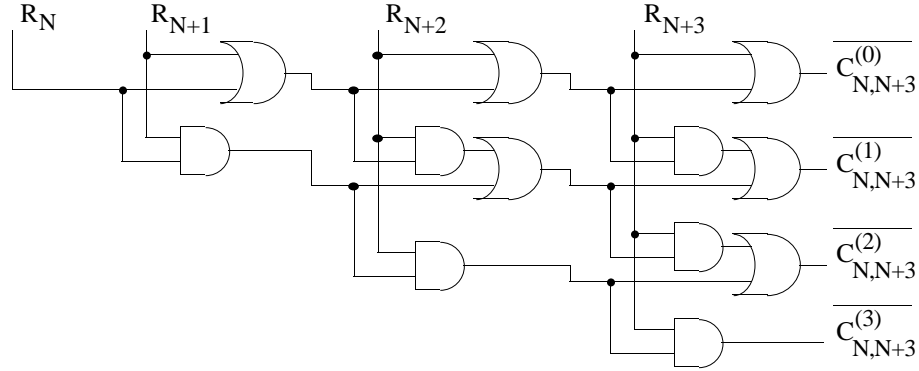
of ports starting with  $l$  and ending with  $k$ . In other words, the equation says that port  $i$  will be free after the group of instructions  $n, n+1, \dots, m-1, m$  have been checked if the port was free before ( $F_{i,n}$ ) and either 1) there were no requests, 2) there was at most one request and at least one port free among the group of  $i-l$  preceding ports, ...,  $i-1$ ) there were  $i-l$  or fewer requests and at least  $i-l$  free ports.

For example, the equations for four ports and a window of four instructions are

$$\begin{aligned}
G_{n,1}^{(1)} &= F_{1,n} & G_{n,2}^{(1)} &= F_{1,n} + F_{2,n} & G_{n,3}^{(1)} &= F_{1,n} + F_{2,n} + F_{3,n} \\
G_{n,2}^{(2)} &= F_{1,n}F_{2,n} & G_{n,3}^{(2)} &= F_{1,n}F_{2,n} + F_{1,n}F_{3,n} + F_{2,n}F_{3,n} \\
G_{n,3}^{(3)} &= F_{1,n} \cdot F_{2,n} \cdot F_{3,n} \\
C_{n,n+3}^{(0)} &= \overline{R_{1,n}} \cdot \overline{R_{1,n+1}} \cdot \overline{R_{1,n+2}} \cdot \overline{R_{1,n+3}} \\
C_{n,n+3}^{(1)} &= \overline{R_{1,n}} \cdot \overline{R_{1,n+1}} \cdot \overline{R_{1,n+2}} + \overline{R_{1,n}} \cdot \overline{R_{1,n+1}} \cdot \overline{R_{1,n+3}} + \overline{R_{1,n+1}} \cdot \overline{R_{1,n+2}} \cdot \overline{R_{1,n+3}} \\
C_{n,n+3}^{(2)} &= \overline{R_{1,n}} \cdot \overline{R_{1,n+1}} \cdot \overline{R_{1,n+2}} + \overline{R_{1,n}} \cdot \overline{R_{1,n+3}} + \overline{R_{1,n+1}} \cdot \overline{R_{1,n+2}} + \overline{R_{1,n+1}} \cdot \overline{R_{1,n+3}} + \overline{R_{1,n+2}} \cdot \overline{R_{1,n+3}} \\
C_{n,n+3}^{(3)} &= \overline{R_{1,n}} + \overline{R_{1,n+1}} + \overline{R_{1,n+2}} + \overline{R_{1,n+3}} \\
F_{1,n+4} &= F_{1,n}(C_{n,n+3}^{(0)}) \\
F_{2,n+4} &= F_{2,n}(C_{n,n+3}^{(0)} + C_{n,n+3}^{(1)}G_{n,1}^{(1)}) \\
F_{3,n+4} &= F_{3,n}(C_{n,n+3}^{(0)} + C_{n,n+3}^{(1)}G_{n,2}^{(1)} + C_{n,n+3}^{(2)}G_{n,2}^{(2)}) \\
F_{4,n+4} &= F_{4,n}(C_{n,n+3}^{(0)} + C_{n,n+3}^{(1)}G_{n,3}^{(1)} + C_{n,n+3}^{(2)}G_{n,3}^{(2)} + C_{n,n+3}^{(3)}G_{n,3}^{(3)})
\end{aligned}$$

In general, the number of instructions to consider is much larger than the number of ports to assign. In a typical case, there are eight instructions in the window and two or three ports. Thus, our strategy is to compute the vector  $\mathbf{C}$  in each of the stream windows and then propagate the  $\mathbf{F}$  vector using equation (6.2).

Fortunately,  $\mathbf{C}$  can be computed as the requests are generated using the counter circuit shown in Figure 6.4. The counter circuit is a variation of the tally circuit used in nMOS cells [mead80,mukh86]. Each row  $j$  of the circuit becomes 1 if there are at least  $j$  requests. Thus,  $\overline{\mathbf{C}}$  is generated. By exchanging AND with OR gates, the matrix  $\mathbf{C}$  can be computed directly. A CMOS implementation can also take advantage of propagate/kill schemes, using a combination of standard gates with transmission gates [mukh86].



**Figure 6.4.** Wavefront computation of the  $\mathbf{C}$  vector.

Using this strategy, we can reduce the critical path of the arbitration for four stream windows to less than the timing for two stream windows using propagation logic. The timing is as follows:

- 1) The matrix  $\mathbf{C}$  for each window is computed in parallel. Also, the output vector  $\mathbf{F}$  for the highest priority window is computed in parallel. This takes about  $W+U$  gate delays, where  $W$  is the size of the window and  $U$  is the number of ports.
- 2) The output vector  $\mathbf{F}$  of the second window is computed from the output  $\mathbf{F}$  of the first window using equation (6.1). This takes two gate delays for a typical case.
- 3) The output  $\mathbf{F}$  of the third window is computed from the output  $\mathbf{F}$  of the second window, as in 2).
- 4) The use matrix  $\mathbf{U}$  for the four window is computed from the output  $\mathbf{F}$  of the third window and the internal matrix  $\mathbf{C}$ . This takes about two to three gate delays.

The circuit of Figure 6.4 can be used for other purposes as well. For example, the scheduler could select two streams for arbitration of some port type if the streams have the highest number of requests for the port type. In this way, the scheduler is giving more chance to those processes that need more resources.

## 6.2 Multithreaded Cache

The speed gap between the processor cycle time and memory access time has continued to grow in recent years. Thus, processors rely on caches to reduce the penalty of memory accesses. However, the performance of a processor is severely reduced by the penalty imposed by cache misses.

The penalty of a cache miss can be reduced by dynamically issuing instructions out-of-order. In a superscalar processor, a data cache miss does not necessarily stop issue of instructions until the cache missed data is delivered from memory. It may stop only issue of the instructions that have data dependency on the cache missed *load* instruction, but the processor can issue out-of-order subsequent instructions. Also, the processor does not need to wait for *stores* that produce a cache miss. However, if the instruction-level parallelism is low as suggested by some authors [joup89], it is very likely that many subsequent instructions will depend on the stalled one therefore effectively stopping the processor until the data comes from memory. Thus, the effect of cache misses on performance is more significant for a superscalar processor because the number of instructions that might have executed if a cache miss did not occur is proportional to the machine parallelism.

A multistreamed superscalar processor running independent instructions streams has more availability of ready-to-issue instructions and therefore does not have to stall on a cache miss; instruction streams that have not missed may continue accessing the cache. The miss rate for a multistreamed architecture may be higher than the corresponding uniprocessor miss rate. However, if the cache design is lock-up free [chua90, krof81], then the cache allows simultaneous accesses. For example, the cache can service a line refill scheduled by a cache miss, while allowing access in parallel to different cache blocks. While the multithreaded miss rate could be higher than the single-stream miss rate, the performance can be higher, as well. This explains why the miss rate is no longer the most important performance metric in multithreaded caches; thus, we must look for alternative metrics.

Hill classifies the misses in uniprocessor caches in four categories: conflict, compulsory, capacity, and coherence [hill87]. Conflict misses are misses that would not occur if the cache was fully-associative and had a true least-recently-used (LRU) replacement policy. Compulsory misses are required in any cache organization because they are the first references to an instruction or piece of data. Capacity misses occur when the cache size is

not sufficient to hold data between references. Coherence misses are misses that occur as a result of invalidation to preserve multiprocessor cache consistency.

Similarly, Agarwal [agar92] classifies the misses experienced by an application in four groups: start-up effects, non-stationary behavior, intrinsic interference, and extrinsic interference. Start-up is caused by initial execution of the program. Changes in the program's working set cause nonstationary behavior misses. The size of the cache causes intrinsic interference, i.e. cache lines owned by an application must be replaced because they conflict. Multiprogramming causes extrinsic misses. Only the number of misses due to interference (intrinsic and extrinsic) increases with a larger number of contexts in a multithreaded architecture.

Sharing the cache has several advantages for multithreaded workloads:

- The need for maintaining duplicated information is reduced, thus the combined working set of the multithreaded cache is smaller than the corresponding single-thread caches.
- The potential ping-pong problems of false sharing in a shared-memory multiprocessor are eliminated. The ping-pong problem occurs when the pattern of access of several processors causes a cache set to be invalidated in each cache periodically, as each cache tries to get ownership of some line. False sharing occurs when the data that processors access are not really shared but map to the same cache line. This problem is specially severe for direct mapped caches.
- The cache utilization is improved because the cache can serve a miss while the cache is being accessed by other threads. This assumes a lock-up free cache design. While the miss rate of a multithreaded cache may be higher, the overall performance may be higher.

On the other hand, there are severe disadvantages of sharing the cache:

- The software design goals may be diametrically opposed for shared or private caches. Sharing the cache encourages the use of false sharing to reduce miss rate. False sharing should be avoided with private caches, even if special purpose cache coherence hardware exists. This could lead to software portability problems for multiprocessors that use combinations of shared/private cache.
- Sharing could lead to a very high extrinsic and intrinsic interference miss rate. The higher miss rate could offset any performance gain of multithreading, leading to contention and possible saturation of the bus used to service the misses. The

problem can be so severe that the multithreaded performance could be lower than the single stream performance.

- Lock-up-free cache designs are more complex since there are simultaneous cache accesses, all in parallel with line refills caused by misses. There is contention for use of resources; conflict resolution hardware may increase the cycle time. This imposes restrictions in the number of simultaneous accesses allowed.

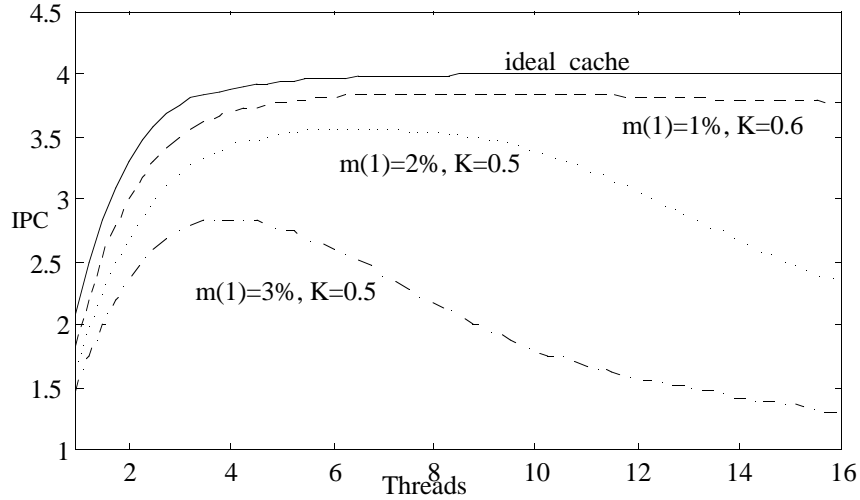
We see that the optimal cache configuration depends on many factors, including the workload. This section analyzes the cache and discusses some ways to reduce the miss rate and/or bus traffic in a shared cache. While it is important to explore many tradeoffs of shared/private caches, we concentrate on ways to reduce the problems of shared caches.

Finally, we discuss implementation technology for the possible use of an on-chip shared primary cache. The expected size of the on-chip caches depends on the implementation technology, with higher-speed technologies generally resulting in smaller on-chip caches. Large caches may be more adequate than small caches for multithread sharing. For example, quite large on-chip caches should be feasible in CMOS but only small caches are feasible in the near term for GaAs/bipolar processors. Thus, although GaAs and bipolar are faster, the higher miss rate of multithreading from their smaller caches could decrease the actual system performance compared to CMOS machines; their performance could be dominated by the speed of the bus communicating with the secondary cache or main memory.

### 6.2.1 Understanding Cache Degradation

A simple cache model helps to understand the adverse effects of increasing the number of contexts on the cache miss ratio. Assume that each context uses a fixed  $1/N$  partition of the total cache, instead of sharing the whole cache between the contexts. Therefore, the total cache size is constant and each context will have a smaller cache for its own use as the number of contexts increases. In reality, contexts may interfere constructively or destructively on shared data. If they do not interfere at all, they might fit in the cache with unequal partitions.

Empirical observations show that the miss ratio  $m$  as a function of the size of the cache  $S$  can be approximated by  $m = AS^{-K}$ , where  $A$  and  $K$  are positive constants that depend on the



**Figure 6.5.** Simulation of the cache with a model. The vertical axis shows the performance in number of instructions per cycle (IPC).

workload [thie89]. The above miss ratio formula appears to be valid for caches between 1K and 256K bytes under uniprocessor execution. An expression for the miss ratio  $m$  with  $N$  contexts each using a cache of size  $S/N$  has been proposed by Saavedra-Barrera [saav90]:

$$m(N) = A(S/N)^{-k} = (AS^{-k})N^k = m(1)N^k, \quad \text{for } N \leq \lfloor m(1)^{-1/K} \rfloor$$

where  $m(1)$  is the miss ratio for one thread.  $K$  is the cache degradation factor. Results reported for three applications show  $K$  ranging from 0.1 to 0.6, with higher  $K$  corresponding to lower initial miss ratio  $m(1)$ . The model of cache degradation due to multithreading shows that returns diminish as the number of threads becomes large.

Figure 6.5 shows the result of a simulation for a particular benchmark and configuration of four functional units using this model for three different values of  $m(1)$  and  $K$ . The results for the ideal cache are shown as the point of reference. As shown, the performance increases until it reaches a maximum after which the performance starts to decline because of a higher miss rate. For some benchmarks with higher miss ratio and/or degradation factors, the maximum that can be obtained is lower with a rapid decrease in performance as the number of threads increase.

From this simulation, it is evident that multithreaded caches can only be used to hold a small number of contexts. The optimal number of streams depends on factors such as cache size, characteristics of the workload, to name a few. For most workloads, the number of threads that can share the cache is very reduced.

## 6.2.2 Cache Behavior for a Small Number of Contexts

We present a study of the multithreaded cache behavior for a small number of contexts, namely one, two, and four. We chose *doduc*, *fpppp*, *eqntott*, *espresso*, *spice*, and *xlisp* from the SPEC'89 benchmark suit. We ran multiple copies of the benchmarks; thus, each stream has an independent data segment, while all the streams share the code segment. This is a worse case for the data cache behavior since the data accesses can only interfere with each other.

We ran simulations for the data cache. To isolate data cache effects, an infinite functional unit configuration and a perfect instruction cache are employed. Thus, performance (IPC) is limited only by data dependencies and cache behavior. The data cache is connected to the main memory through a data bus used to service cache refills. Cache refills are non-split transactions that take 10 cycles. We assume a perfect lock-up free data cache design. The line size is 32 bytes, the cache is write-back and the replacement policy is LRU.

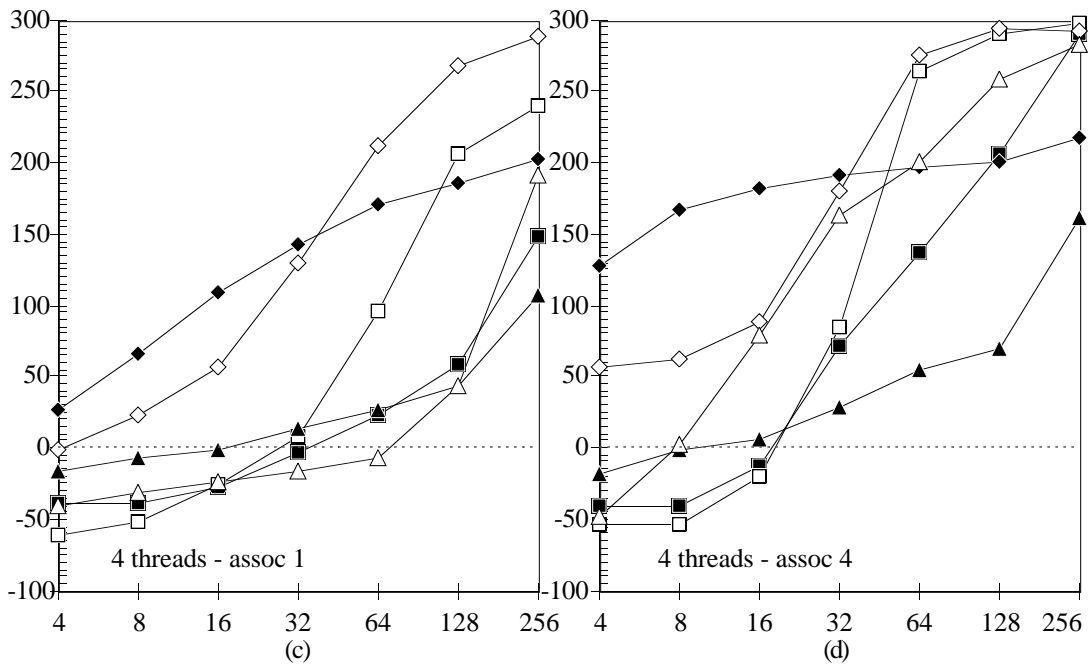
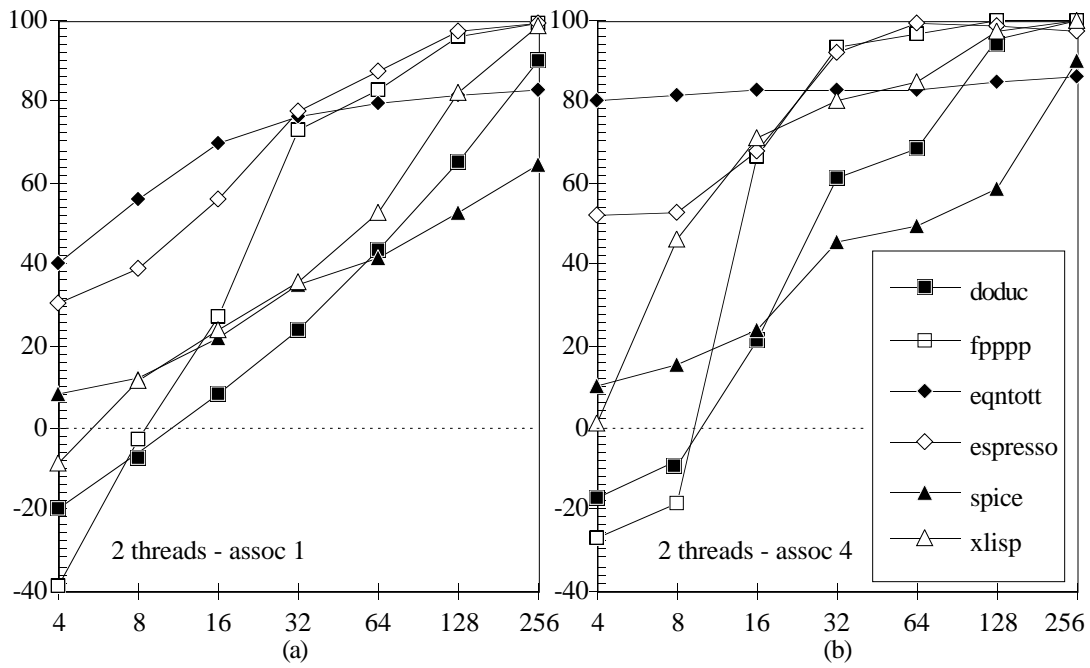
The number of simultaneously active working sets in a multithreaded architecture is higher than in a single thread cache. Thus, an increase in the associativity of the cache reduces the conflicts, both intrinsic (within the thread) and extrinsic (between different threads). Figure 6.6 shows the simulation results obtained for cache associativities one (direct-mapped), and four.

Since the miss ratio is not truly indicative of multithreaded processor behavior, we chose the relative improvement of  $N$  threads over one thread as our performance measure. The relative performance is defined as

$$R(N) = (IPC_N - IPC_1) / IPC_1$$

where  $IPC$  is the number of instructions-per-cycle, and  $N$  is the number of threads, i.e. two or four. Therefore, the maximum relative performance is 100% and 300% for two and four threads, respectively, since there are multiple copies of the same program with different data sets in the data cache. Note that the relative performance could be negative indicating worse performance than a single-thread.

Figure 6.6 shows results for two (a,b) and four threads (c,d), and cache associativities one (a,c), and four (b,d). We chose 4, 8, 16, 32, 64, and 256 K bytes cache sizes as they represent the range of primary cache sizes that can be found on existing processors.



**Figure 6.6.** Relative improvement over one thread for two (a,b) and four threads (c,d), for caches with associativity one (a,c), and four (b,d).

Our simulation shows very distinct patterns of cache behavior. *Doduc* and *fpppp* are benchmarks that exhibit bad cache behavior (negative improvement) for small caches for two threads, and even worse for four threads. Increasing the associativity does not affect performance for small caches (4, 8, and 16 Kilobytes), but the improvement becomes very significant for relatively large caches. *Eqntott* performance is acceptable for small caches; its performance improves with increasing associativity but does not reach the maximum predicted (100% and 300 %); there is a secondary effect that reduces performance growth. *Spice* has an relatively acceptable small cache performance with a slow growth with the cache size. *Spice* performance is relatively unaffected by increasing associativity (except for 256 K). *Espresso* has a relative good performance for small caches and improves significantly with increasing cache size and associativity. Similarly, the performance of *xlisp* improves with the cache size and associativity.

There is a positive correlation between the percentage of memory operations in the instruction mix and the multithreaded cache performance. Table 3.3 in Chapter 3 shows the instruction mixes of the benchmarks. For example, *eqntott* and *espresso* with 25% load/stores have the best performance. Benchmarks with high percentage of load/store (*fpppp* 59%, *doduc* 41% and *xlisp* 41%) have bad performance for small caches; however, an increase in cache size dramatically improves their performance. *Spice* is a benchmark with 33% load/stores with fair performance for small caches; however, its performance increases slowly with the cache size. Thus, the performance of *spice* is determined by the memory access pattern.

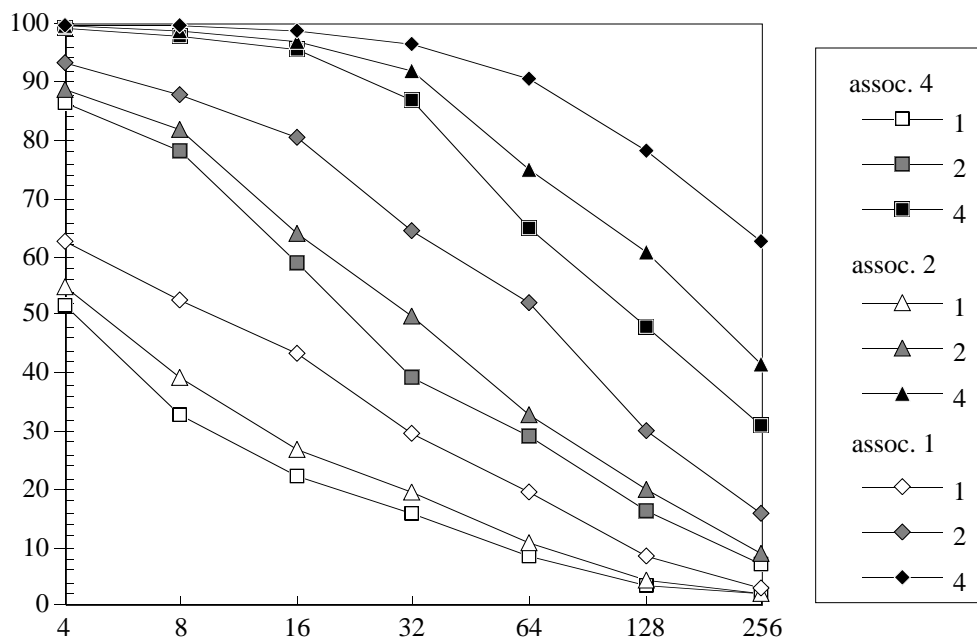
Multithreaded loss of performance is not only due to cache interference but to contention in the bus used to service the refills. Figure 6.7 shows the average bus utilization for the six benchmarks, for two/four threads and associativities 1, 2, and 4. As shown in the figure, the bus is saturated (100% utilization) for four threads and small caches. Bus saturation is highly undesirable because the average memory latency for a refill increases significantly when the bus is almost saturated. The situation is even worse if we consider the bus traffic generated by instruction cache which we have assumed ideal.

As shown in Figure 6.7, increasing associativity has the predictable effect of steadily reducing the bus utilization. We can see that for one and two threads the reduction from changing the associativity one to two is much greater than from two to four, indicating that performance improvement will not grow beyond some given associativity value. For four

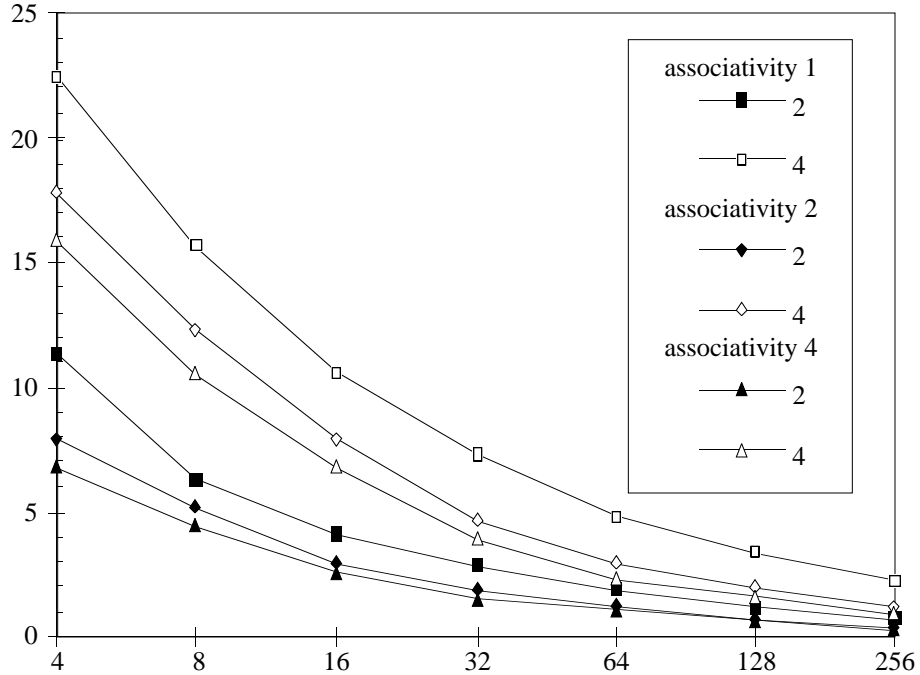
threads the situation is different since there is more need for higher associativity for the increasing number of working sets.

Figure 6.8 shows the average extrinsic interference (between threads) for cache associativities 1, 2, and 4, and two and four threads. The extrinsic interference is defined as the percentage of misses produced by a thread replacing another thread's cache line. By definition, the interference of the single thread is zero. As expected, increasing the associativity has the steady effect of reducing the interference.

We observe that while increasing associativity helps for most benchmarks, some benchmarks remain relatively unaffected, *spice*, for example. Also, there is the severe problem of the negative improvements for small caches; in these cases it is better to have private caches. This suggests that new non-traditional approaches have to be examined to improve multithreaded cache performance.



**Figure 6.7.** Average percentage of bus utilization for multithreaded caches with associativities 1, 2, and 4, and one, two, and four threads. The horizontal axis is the cache size in kilobytes.



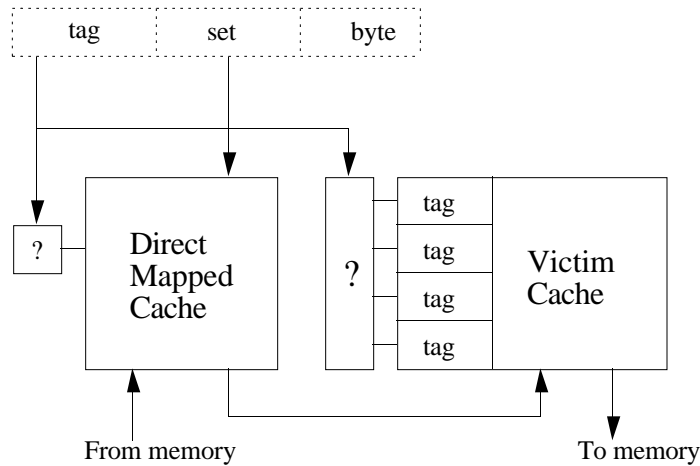
**Figure 6.8.** Average interference for two and four streams and associativities 1, 2, and 4. The horizontal axis is the cache size in kilobytes. The vertical axis is the percentage of cache interference.

### 6.2.3 Victim Cache

Jouppi [joup90] proposed the victim cache as a way to improve the performance of direct mapped caches. Here we examine its performance for multithreaded contexts.

Even though direct-mapped caches have more conflict misses due to their lack of associativity, their performance is still better than set-associative caches when the access time costs for hits are considered. The direct-mapped cache is the only cache configuration where the critical path is merely the time required to access a RAM. The idea is to somehow provide additional associativity without adding to the critical access path for a direct-mapped cache to reduce conflict misses. Conflict misses account for between 20% and 40% of all direct-mapped cache misses [hill87].

Victim caching is an improvement to miss caching [joup90] that loads a small fully-associative cache with the victim of a miss and not the requested line. Victim caching places a small fully-associative cache between a cache and its refill path. The small fully-associative cache is loaded with the victim of a miss.



**Figure 6.9.** Organization of victim cache.

Figure 6.9 shows the organization of our victim cache. The cache is effectively divided in two sections to exploit both the spatial and temporal locality of the memory references. The two halves are accessed in parallel. While the set portion of an address is used to access a large direct mapped cache, the tag is compared (symbol ?) with all the tags of the victim cache. A memory reference hits if either of the two halves contain the corresponding tag. On a miss, two separate replacement policies are activated in both caches. The line is replaced in the direct mapped cache while the displaced line from the direct mapped cache goes to the victim cache where it replaces the least-recently-used (LRU) line. With victim caching, no data line appears both in the direct-mapped cache and the victim cache.

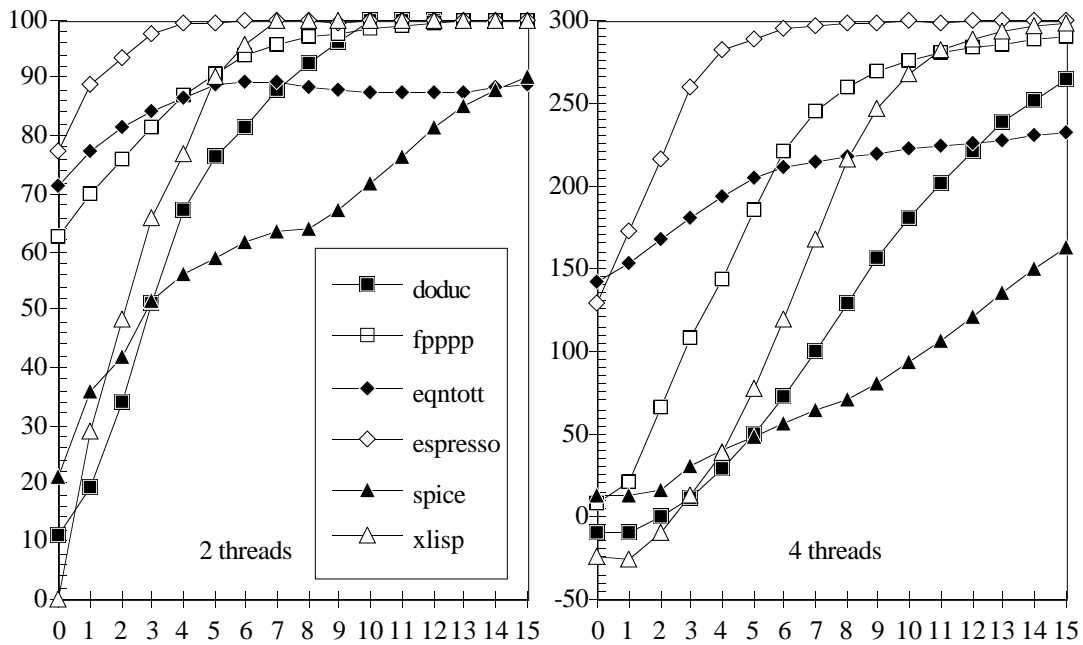
Our victim cache is somehow different to the architecture proposed by Jouppi; in its approach in the case of a miss in the direct-mapped cache that hits in the victim cache, the contents of the direct-mapped cache line and the matching victim cache line are swapped. In our approach the cache line remains in the victim cache until it gets displaced because it is the oldest reference. Our approach exploits the victim cache for both temporal locality and store buffering.

The function of the victim cache is similar to the function of the *store buffers* already employed in architectures such as the MPC601 [moto93]. The store buffers are used to store cache lines invalidated by the cache coherence detector mechanism that monitor a snoopy bus. Victim caching reuses the store buffers for cache access. Thus, victim caching can be seen as an extension to the use of store buffers.

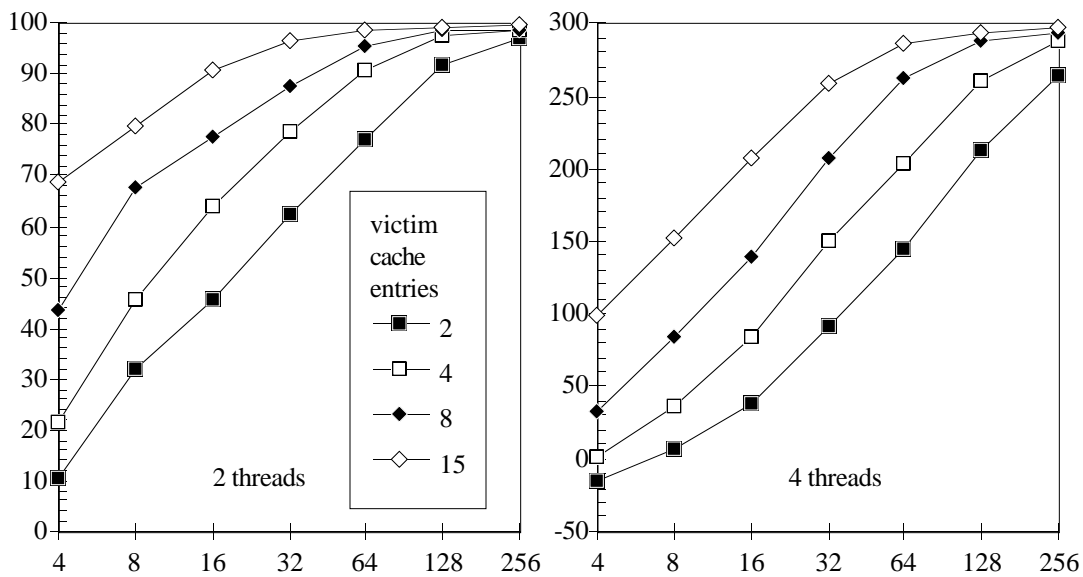
For example, the MPC601 has a memory unit with a write queue for buffering. Some entries of the queue are dedicated to writing cache sectors to system memory after a modified sector is hit by a snoop from another processor or snooping device on the system bus, to guarantee that a high-priority operation receives a deterministic response when snooping hits a modified sector; these entries obviously do not contain valid information in the victim cache. The other entries of the queue which contain valid information are used in a conventional way to store written back modified (dirty) lines that have been deallocated; that is, when a cache sector is full, the least recently used cache line is deallocated by first being copied into the write queue and from there to system memory if it is modified. In the victim cache, both dirty and non-dirty displaced elements are stored. Note that snooping can occur after a sector has been pushed out into the write queue and before the data has been written to system memory. Therefore, to maintain a coherent memory, the tags in the victim cache are compared to snooped addresses in the same way as the cache tags. If a snoop hits an entry the data are first stored in system memory before they can be loaded into the cache of the snooping bus master.

Figure 6.10 shows the performance of our six benchmarks for a direct mapped cache of size 32K as a function of the number of entries in the victim cache. A data cache size of 32K is typical of processors such as the RS6000 [groh90]. Once again we observe different patterns for the benchmarks. We can see that a victim cache with 15 entries almost completely masks the interference effects for two threads; the average improvement is close to 96%. For four threads the 15-entry victim cache brings a good improvement; the only exception is *spice*, with a slow growth. We again observe the odd behavior of *eqntott*, with secondary effects that level off performance followed by a secondary improvement. A few entries in the victim cache bring acceptable performance: six entries for two threads and twelve entries for four threads.

Figure 6.11 shows the average performance improvement of the victim cache as a function of the cache size. The figure shows results for 2, 4, 8, and 15 entries in the victim cache used with direct-mapped data caches of different sizes. In spite of the dissimilar behavior of the benchmarks, the figure shows the regular pattern of improvement of adding more entries to the victim cache. In general, smaller caches benefit the most from the victim cache since the improvement is almost linear with the number of entries. For mid-range caches (32K), the improvement is almost logarithmic, i.e. doubling the number of entries produces the same increase, approximately.



**Figure 6.10.** Percentage of improvement in performance for a 32K direct mapped cache as a function of the number of entries in a victim cache. The left figure is for two threads while the right one is for four threads.



**Figure 6.11.** Average percentage of improvement as a function of the cache size (in kilobytes), for 2, 4, 8, and 15 entries in the victim cache. The left figure is for two threads while the right figure is for four threads.

Table 6.1 provides a performance comparison between the effect of the victim cache and the effect of increasing the associativity in a conventional cache. The table shows the average number of entries in the victim cache needed to produce the same or better IPC than the effect of increasing the associativity in a conventional cache. As seen from the table, the number of entries in the victim cache is small to produce the same effect. This is specially true for small caches where the victim cache has more effect than the associativity of the cache, because the size of the cache cannot simultaneously contain the whole working set of all the threads. For large caches, there is not much difference between the victim cache and the associativity since both have the same effect of reducing the number of conflicts in the cache. We see that a victim cache with four entries provides equal or better performance than a corresponding cache with associativity four. Furthermore, a victim cache with eight entries can annihilate any performance degradation produced by multithreaded caches holding up to four contexts.

cache size	Associativity 2			Associativity 4		
	1 stream	2 streams	4 streams	1 stream	2 streams	4 streams
4	1.7	1.8	2.5	2.5	2.5	3.8
8	2.2	1.8	2.7	3.0	2.5	3.8
16	2.2	2.7	3.0	3.2	2.8	3.7
32	1.5	2.3	2.5	1.8	2.8	3.2
64	1.5	2.0	2.3	1.7	2.2	3.2
128	1.2	1.5	1.8	1.2	2.0	2.2
256	1.0	1.3	1.5	1.0	1.5	2.0

**Table 6.1.** Average number of entries in the victim cache to produce the same effect as increasing the cache associativity for one, two and four streams.

Victim caches can be useful as second-level caches, especially for small caches where capacity misses are important. As the size of the cache increases, the percentage of the conflict/compulsory misses is more important than capacity misses. Since the number of conflict misses increases with increasing line sizes, the large line sizes of second-level caches also tend to increase the potential usefulness of victim caches.

A victim cache can contain many lines that conflict not only at the first level but also at the second level. Thus, a first-level victim cache can also reduce the number of conflict misses at the second level [joup90]●

## 7 Conclusion

Recent superscalar designs use several functional units [whit93]. As the trend moves towards integrating more functional units within the processor, designers face the challenge of utilizing these additional resources effectively. Superscalar processors face limitations to exploit the additional functional units due to the inherent limit of instruction-level parallelism within a single instruction stream. Multistreaming provides an alternative way to improve the utilization of the superscalar processor with multiple functional units.

A multistreamed processor is provided with hardware support for a limited number of streams. Each stream uses a context frame, which is the collection of all the registers needed to hold the context of a program, such as the program counter, register file, etc. Therefore, a limited number of threads can run simultaneously in the multistreamed processor.

Recent operating systems and parallelizing compilers provide multiple threads of computation. Integrating multistreaming within superscalar architectures is an effective method for maximally exploiting the additional functional units needed for this new generation of software.

### 7.1 General Architecture

Chapter 2 presented a general framework for the study of multithreaded, multistreamed, superscalar processors. We presented a classification of multithreading, namely coarse-grain and fine-grain, according to the number of streams present in the processor. A stream is defined as the hardware support needed to hold the running context of several threads (activation), thus a stream has its own copy of register files (integer and floating-point) and other special registers. Coarse-grain (blocked) multithreading uses one stream, and is used mainly to hide long-latency memory operations. Fine-grain (interleaved) using several streams (multistreamed) can improve both tolerance to memory latency and functional unit utilization of the superscalar processor. However, interleaving implies a higher cost in terms of additional scheduling hardware and duplicated register files.

We also presented general architecture principles for building a multistreamed processor using higher circuit integration. A processor with a higher clock rate is easier to implement if global interconnections are minimized using the principle of locality of the functions, namely the minimization of the communication among the different building blocks of the processor.

## **7.2 Performance Evaluation**

Chapter 3 presented a performance evaluation tool for Dynamic Superscalar Multistreamed Processors (DSMP).

As more functional units are added within the processor, DSMP show that they can make much more efficient use of the resources than single streamed, superscalar architectures. Our simulation results for a random schedule and single memory latency show performance increases ranging from 31.5 percent for a base configuration to 284 percent for a configuration with many functional units. DSMP processors provide good tolerance to memory latency. Our studies also show little impact of the schedule policy on the overall performance, but large impact on the individual streams. Results show that multistreamed superscalar architectures can achieve high processor efficiency given the constraints of functional units, dependencies, and issue window size. Small window sizes can be used to obtain adequate performance.

## **7.3 Performance Estimation**

Chapter 4 presented an analytic technique for evaluating the performance of multistreamed, superscalar processors. Our results demonstrate that the technique produces accurate instructions-per-cycle (IPC) estimates of the overall performance through comparison with a multistreamed RS/6000 simulator.

The analytical technique provides a quick way to examine the many variations of an architecture at a high level of abstraction. The technique only requires simple descriptions of the workload and architectural configuration as input parameters. These parameters are easily measured or estimated using tools that are commonly available. In addition, the simplicity of the model makes it easy to implement and much faster to execute than a hardware simulator.

Performance estimation is separated into two major parts: 1) model of structural hazards, and 2) model of control and data hazards. The rationale behind this division is that the modeling of structural hazards is primarily dependent on the architectural (hardware) configuration while the modeling of control and data hazards is primarily dependent on the workload. We employ a Markov Chain for the model of structural hazards. Furthermore, we split the Markov Chain modeling all the structural hazards in smaller subchains to reduce the complexity associated with solving a large Markov Chain. Control and data hazards are characterized by a histogram vector obtained from simulations for a single stream.

We presented analytic expressions for the saturation value for different schedules. The saturation value is the maximum performance that can be achieved, given the characteristics of the workload, the configuration in terms of functional units, and the characteristics of the schedule.

Each instruction stream is interpreted as a random stream of instructions of each type. The accuracy of the model depends on the assumptions that 1) the instruction mix is a stationary process, and 2) instructions appear in the instruction stream in a random manner. We presented two estimators ( $\sigma, \delta$ ) to measure the degree of closeness to these assumptions:

## **7.4 Multistreamed Instruction Issue Mechanism**

Chapter 5 presented an issue scheme that resolves dependencies in a way which can be considered intermediate between the Tomasulo *after-issue* algorithm and the Dispatch stack *before-issue* scheme. The paper exposes some of the tradeoffs for dependency resolution with logic located in different places of the processor. Communication between the different components constitutes an important bottleneck in a design of a superscalar processor. Thus, the communication needed to keep the scheme is reduced by the use of comparative logic which can be considered an extension of ordinary match logic.

The design uses a distributed version of the reorder buffer because its functions are assumed by the issue window in the control unit, and the result buffer in the data units. The result buffer supports a limited form of register renaming which can be improved with additional logic that keeps track of WAW hazards. Another objective is to keep the logic simple, regular, and amenable to VLSI layout to avoid long critical delay paths that reduce the maximum achievable clock rate. For example, functional units do not have freeze logic.

Several authors treat speculative execution, precise interrupts, and instruction issue separately. However, these aspects are interrelated, so that the methodology treats them in a simple, unified manner. The chapter mainly discusses the implementation of precise interrupts at the finest granularity. However, there is a brief discussion of some of the tradeoffs that can be done which reduce the hardware complexity at the cost of reducing the granularity of interrupts.

## **7.5 Performance Tradeoffs**

We discuss in Section 6.1 the problems of sharing functional units. In particular, we concentrate on reducing the overhead caused by arbitration of functional units in a multistreamed environment. Our scheme is designed for a small number of functional units and streams. A high number of functional units and/or streams needs a more complex interconnection network that probably is not cost effective because of the increased complexity. In that case it would be better for each stream to share a reduced number of functional units.. Only functional units with high implementation cost (area, for example) could be shared by all the streams.

Section 6.2 discusses some of the problems of sharing a cache by multiple contexts. In spite of the higher miss rate of multistreaming, the performance can be higher as well. Instead of using the miss rate, we use the relative performance IPC improvement as our performance metric. Small caches are probably not adequate for sharing because of the working set size of multiple threads. Bigger caches, which experience more conflict than capacity misses, are more amenable for sharing. We also show the improvement of adding a small victim cache to exploit both spatial and temporal locality of the memory references. Victim caches are shown more effective in improving performance than a corresponding increase in the cache associativity.

## BIBLIOGRAPHY

- [agar90] A. Agarwal, "APRIL: A Processor Architecture for Multiprocessing," *Proceedings 17th Symposium on Computer Architecture*, May 1990, pp. 104-114.
- [alve90] G. Alverson, R. Alverson, B. Koblenz, D. Callahan, A. Porterfield, B. Smith, "Exploiting Heterogeneous Parallelism on a Multithreaded Multiprocessor," Tera Computer Company. *Workshop on Multithreading, Supercomputing'91*, October 1991.
- [agar92] A. Agarwal, "Performance Tradeoffs in Multithreaded Processors," *IEEE Transactions on Computers*, Vol. 3, No. 5, Sept. 1992, pp. 525-539.
- [arvi88] Arvind, D.E. Culler, K. Ekanadham, "The Price of Asynchronous Parallelism: An Analysis of Dataflow Architectures," Computation Structures Group Memo 278, June 1988. Massachusetts Institute of Technology.
- [aust90] T.M. Austin, G.S. Sohi, "Dynamic Dependency Analysis of Ordinary Programs," *Proceedings 19th Symposium on Computer Architecture*, May 1992, pp. 342-351
- [bako90] H.B. Bakoglu, *Circuits, Interconnections, and Packaging for VLSI*. Addison-Wesley Publishing Company, 1990.
- [bose84] R.C. Bose, B. Manvel, *Introduction to Combinatorial Theory*, Wiley Series in Probability and Mathematical Statistics, 1984.
- [butl92] M. Butler, T-Y Yeh, Y. Patt, "Single Instruction Stream Parallelism is Greater than Two," *Proceedings 18th Symposium on Computer Architecture*, May 1992, pp. 276-286.
- [chua90] S. C-M. Chuang, A. Bruss, "Architecture and Design of a Pseudo Two-port VLSI Snoopy Cache Memory," *IEEE COMPUERO*, 1990, pp. 400-407.
- [cont92] T.M. Conte, "Systematic Computer Architecture Prototyping," PhD Thesis, Electrical Engineering, University of Illinois at Urbana-Champaign, 1992.
- [cour77] P.J. Courtois, *Decomposability. Queuing and Computer System Applications*. Academic Press. 1977.
- [cull90] D.E. Culler, G.M. Papadopoulos, "The Explicit Token Store," *Journal of Parallel and Distributed Computing*, Dec. 1990, pp. 289-308.
- [cull91] D.E. Culler, A. Sah, K.E. Schauer, T. Eicken, J. Wawrzynek, "Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine," *Proc. 5th International Conference Architectural Support for Programming Languages and Operating Systems*, 1991, pp. 164-175.
- [cull92] D.E. Culler, M. Gunter, J.C. Lee, "Analysis of Multithreaded Microprocessor under Multiprogramming", Technical report no. UCB/CSD 92/687. May 1992. University of California, Berkeley.
- [dadd91] G.E. Daddis Jr., H.C. Torng, "The Concurrent Execution of Multiple Instruction Streams on Superscalar Processors," *Proceedings 1991 Conference on Parallel Processing*, Vol. I, pp. 76-83.

- [dwy87] H. Dwyer, H.C. Torng, "A Fast Instruction Dispatch Unit for Multiple and Out-of-Sequence Issuances," School of Electrical Engineering Technical Report EE-CEG-87-15, November 1987, Cornell University, Ithaca, N.Y.
- [dwy91] H. III Dwyer, "A Multiple, Out-of-Order, Instruction Issuing System for Superscalar Processors," PhD dissertation Cornell University, 1991. University Microfilms International, No. 9204017.
- [denn88] J.B. Dennis, G.R. Gao, "An efficient pipelined dataflow processor architecture," *Proceedings of the Supercomputing '88 Conference*, Florida, Nov. 1988, pp. 368-373.
- [digi92] Digital Equipment Corporation. DECChip 21064-AA RISC Microprocessor Preliminary Data Sheet. April 29, 1992.
- [dodd92] J.M. Dodd, "Design and Analysis of SPArTAn: A Scientific Processor Architecture for Task-Partitioned Applications," ECE Tech. report #92-12, University of California, Santa Barbara, 1992.
- [feld91] S.I. Feldman, D.M. Gay, M.W. Maimone, and N.L. Schryer, "A Fortran-to-C Converter," Computing Science Technical Report No. 149, AT&T Bell Laboratories, Murray Hill, NJ, 1991.
- [flyn70] M.J. Flynn, A. Podvin, and K. Shimizuk, "A Multiple Instruction Stream processor with shared resources," *Parallel Processor System*, Washington D.C., Spartan, 1970.
- [fish81] J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, Vol. C-30, No. 7, July 1981, pp. 478-490.
- [fish84] J.A. Fisher, "The VLIW Machine: A Multiprocessor for Compiling Scientific Code," *IEEE Computer*, July 1984, pp. 45-53.
- [groh90] G.F. Grohoski, "Machine Organization of the IBM RISC System/6000 processor," *IBM Journal of Research and Development*, Vol. 34, No. 1, January 1990, pp. 37-58.
- [hall91] B.C. Hall, K. O'Brien, "Performance characteristics of architectural features of the IBM RISC System/6000. *ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, 1991, pp.303-309.
- [hals88] R.H. Halstead Jr., T. Fujita, "MASA: Multithreaded Processor Architecture for Parallel Symbolic Computing," *Proceedings 15th Symposium on Computer Architecture*, Honolulu, Hawaii, June 1988, pp. 443-451.
- [hara92] H. Hara, T. Sakurai, T. Nagamatsu, K. Seta, H. Momose, Y. Niitsu, H. Miyakawa, K. Matsuda, Y. Watanabe, F. Sano, A. Chiba, "0.5-um 3.3-V BiCMOS Standard Cells with 32-kilobyte Cache and Ten-Port Register File", *IEEE Journal of Solid-State Circuits*, Vol. 27, No. 11, Nov. 1992, pp. 1579-1582.
- [henn90] J.L. Hennessy, D.A. Patterson, *Computer Architecture, A Quantitative Approach*, Morgan Kaufmann Publishers, 1990.
- [hill87] M. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. Ph.D. Thesis, University of California, Berkeley, 1987.

- [hida93] Y. Hidaka, H. Koike, H. Tanaka, "Multiple Threads in Cyclic Register Windows," *Proceedings of the 20th International Symposium on Computer Architecture*, May 16-19, 1991, San Diego, California, pp. 131-142.
- [hira92] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, T. Nishizawa, "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads," *Proceedings 19th Symposium on Computer Architecture*, May 1992, pp. 136-145.
- [hwu87] W.W. Hwu, Y. Patt, "Checkpoint Repair for High-Performance Out-of-Order Execution Machines," *IEEE Trans. on Computers*, Vol. C-36, No. 12, Dec. 1987, pp. 1496-1514.
- [ibm92] *IBM AIX Version 3.2 for RISC System/6000 Assembler Language Reference*, IBM Corporation, 1992.
- [john91] M. Johnson, *Superscalar Microprocessor Design*. Prentice Hall, 1991.
- [joup89] N.P. Jouppi, "The Nonuniform Distribution of Instruction-level and Machine Parallelism and its Effect on Performance," *IEEE Transactions on Computers*, Vol. 38, No. 12, Dec. 1989, pp.1645-1658.
- [joup90] N.P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *Proceedings 17th Symposium on Computer Architecture*, May 1990, pp. 364-373
- [kami79] W.J. Kaminsky, E.S. Davidson, "Developing a Multiple-Instruction-Stream Single-Chip Processor," *IEEE Computer Magazine*, Dec. 1979.
- [kato92] T. Kato, K. Suginuma, N. Bagherzadeh, "On Design and Performance Analysis of a Superscalar Architecture," *1992 Int. Conf. on Parallel Processing*, Vol. I. pp. 171-178
- [kowa85] J.S. Kowalik, ed., *Parallel MIMD Computation: HEP Supercomputer and its Applications*, MIT Press, 1985.
- [kowa89] L. Kohn, S.-W. Fu, "A 1,000,000 Transistor Microprocessor," *1989 IEEE International Solid-State Circuits Conference*, pp. 54-55, 1989.
- [krof81] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," *Proceedings 8th Symposium on Computer Architecture*, 1991, pp. 43-53.
- [leno92] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, J. Hennesy, "The DASH Prototype: Implementation and Performance," *Proceedings 19th International Symposium on Computer Architecture*, May 1993, Gold Coast, Australia, pp. 92-103.
- [mccr91] D.C. McCrackin, "Eliminating Interlocks in Deeply Pipelined Processors by Delay Enforced Multistreaming," *IEEE Transactions on Computers*, Vol. 40, No. 10, October 1991, pp. 1125-1132.
- [mead80] C. Mead, and L. Conway, *Introduction to VLSI Systems*. Addison-Wesley, 1980.
- [mont92] J-M. Monti, G. Gao, "Efficient Interprocessor Synchronization / Communication on a Dataflow Multiprocessor Architecture," *Proceedings 1992 Conference on Parallel Processing*, Vol. I, pp. 220-223.

- [moto93] *PowerPC 601, RISC Microprocessor User's Manual*. Motorola Inc. 1993.
- [mukh86] A. Mukherjee, *Introduction to nMOS and CMOS VLSI systems design*. Prentice-Hall, 1986.
- [nikh89] R.S. Nikhil, Arvind, "Can dataflow subsume von Neumann computing," *Proceedings 16th Symposium on Computer Architecture*, Jerusalem, May 1989, pp. 262-272.
- [nikh92] R.S. Nikhil, G.M. Papadopoulos, Arvind, "T: A Multithreaded Massively Parallel Architecture," *Proc. 19th Symposium Computer Architecture*, May 1992, pp. 156-167.
- [nemi90] M.D. Nemirovsky, "DISC, A Dynamic Instruction Stream Computer," Ph.D. Dissertation, University of California, Santa Barbara, September 1990.
- [nemi91] M.D. Nemirovsky, F. Brewer, R.C. Wood, "DISC: Dynamic Instruction Stream Computer," *Proceedings of the 24th Annual International Symposium on Microarchitecture*, Nov. 1991, Albuquerque, New Mexico, pp. 156-167.
- [oehl91] R.R. Oehler, M.W. Blasgen, "IBM Risc System/6000: Architecture and Performance," *IEEE Micro*, June 1991, pp. 14-17,56-61.
- [oust93] J.K. Ousterhout, *An Introduction to Tcl and Tk*. Addison-Wesley, 1993.
- [park91] W.W. Park, "Performance-Area Trade-offs in a Multithreaded Processing Unit," PhD dissertation, The University of Texas at Austin, 1991. University Microfilms International, No. 9212608.
- [pope91] V. Popescu, M. Schultz, J. Spracklen, G. Gibson, B. Lighthner, D. Isaman, "The Metaflow Architecture," *IEEE Micro*, June 1991, pp. 10-13,63-71.
- [pras91] R.G. Prasad, C.L. Wu, "A Benchmark evaluation of a multi-threaded RISC processor architecture," *Proceedings 1991 International Conference on Parallel Processing*, Vol. I, pp. 84-91, 1991.
- [saav90] R.H. Saavedra-Barrera D.E. Culler, T.V. Eicken, "Analysis of Multithreaded Architectures for Parallel Computing," *ACM 2nd Annual Symposium on Parallel Algorithms and Architectures*, Crete, Greece, July 1990, pp. 169-178.
- [serr93] M.J. Serrano, D. Donalson, M.D. Nemirovsky, R.C. Wood, "DISC: Dynamic Instruction Stream Computer. A performance Evaluation," *Proceedings of the Hawaii Systems and Sciences Conference*, Maui, Hawaii, January 1993, pp. 183-192.
- [serr94] M.J. Serrano, W. Yamamoto, R. Wood, M. Nemirovsky, "A model for Performance Estimation in a Multistreamed Superscalar Processor," *Seven Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, Springer-Verlag, May 1994, Vienna, Austria.
- [shet91] J. Shetler, S.E. Butner, "Multiple Stream Execution on the DART Processor," *Proceedings 1991 International Conference on Parallel Processing*, Vol. I, pp. 92-96.
- [smit81] B.J. Smith, "Architecture and Applications of the HEP multiprocessor computer system," *SPIE 298*, 1981, pp. 241-248.

- [smit88] J.E. Smith, A.R. Pleszkun, "Implementing Precise Interrupts in pipelined processors," *IEEE Transactions on Computers*, Vol. 37, No. 5, May 1988, pp. 692-700.
- [smit90] B. Smith, R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield. "The Tera Computer System," *Proceedings of Supercomputing'90*, pp. 1-6.
- [sohi90] G.S. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers," *IEEE Transactions on Computers*, Vol. 39, No. 3, March 1990, pp. 349-359.
- [souz92] E. de Souza e Silva, P.M. Ochoa, "State Space Exploration in Markov Models," *Performance Evaluation Review*, Vol. 20, No. 1, June 1992, pp. 152-166.
- [stal86] C.A. Staley, "Design and Analysis of the CCMP: A Highly Expandable Shared Memory Parallel Computer," Ph.D. Diss. University of California, Santa Barbara, August 1986.
- [theo92] K.B. Theobald, G.R. Gao, and L.J. Hendren, "On the Limits of Program Parallelism and its Smoothability," *IEEE Micro* 25, 1992, pp. 10-19.
- [thie89] D. Thiebaut, "On the fractal Dimension of Computer Programs and its Application to the Prediction of the Cache Miss Ratio," *IEEE Transactions on Computers*, Vol. 38, No. 7, July 1989, pp. 1012-1026.
- [thor64] J. E. Thornton, "Parallel Operation in the Control Data 6600," *Proceedings Spring Joint Computer Conference*, 1964.
- [toma67] R.M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, Vol. 11, Jan. 1967, pp. 25-33.
- [torn84] H.C. Torng, "An instruction issuing mechanism for a processor having multiple functional units," School of Electrical Engineering, Cornell University, Ithaca, NY. Tech. Rep # EE-CEG-84-1, Feb. 1984.
- [uht92] A.K. Uht, "Concurrency Extraction via Hardware Methods. Executing the Static Instruction Stream," *IEEE Trans. on Computers*, Vol. 41, No. 7, July 1992, pp. 826-841.
- [wang91] L. Wang, C-L Wu, "Distributed Instruction Set Computer Architecture," *IEEE Transactions on Computers*, Vol. 40, No. 8, August 1991, pp. 915-933.
- [webe89] W. Weber, A. Gupta, "Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results," *Proceedings 16th Symposium on Computer Architecture*, May 1989, pp. 273-280.
- [welb93] E. H. Welbon, C.C. Chan-Nui, D.J. Shippy, and D.A. Hicks, "The Power2 Performance Monitor," *IBM RISC System/6000 Technology: Volume II*. Sept. 23, 1993, pp. 14-21.
- [whit93] S.W. White, S. Dhawan. "Power2: Next Generation of the RISC System/6000 Family". *IBM RISC System/6000 Technology: Volume II*, Sept. 23, 1993, pp. 2-13.
- [yama94] W. Yamamoto, M. J. Serrano, R. Wood, M. Nemirovsky, "Performance Estimation of Multistreamed, Superscalar Processors," *Proceedings of the Hawaii Conference on Systems and Science*, Hawaii, January 1994, pp. 195-204.