MEMORY-SYSTEM DESIGN CONSIDERATIONS FOR DYNAMICALLY-SCHEDULED MICROPROCESSORS

by

Keith Istvan Farkas

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Memory-System Design Considerations for Dynamically-Scheduled Microprocessors

Keith Istvan Farkas

Doctor of Philosophy

Graduate Department of Electrical and Computer Engineering

University of Toronto

1997

Dynamically-scheduled processors challenge hardware and software architects to develop designs that balance hardware complexity and compiler technology against performance targets. This dissertation presents a first thorough look at some of the issues introduced by this hardware complexity. The focus of the investigation of these issues is the register file and the other components of the data memory system. These components are: the lockup-free data cache, the stream buffers, and the interface to the lower levels of the memory system.

The investigation is based on software models. These models incorporate the features of a dynamically-scheduled processor that affect the design of the data-memory components. The models represent a balance between accuracy and generality, and are used to identify performance trends and design relationships rather than to evaluate the performance of specific designs.

The models are used to identify and explore the relationships between the following system aspects: the dispatch queue, the physical registers, the type of exceptions, the issue width of the processor and the number of functional units, the lockup-free data cache, the stream buffers, and the interface to the lower levels of the memory system. Implementation-related relationships are exposed through an analysis of how various design decisions impact the cycle time of a register file.

Following from this analysis, an investigation of a novel architecture is conducted. The novelty of this architecture is the partitioning of key components to reduce the cycle time of the processor. A comprehensive examination is given of the software and hardware complexities the partitioning introduces.

Static instruction scheduling algorithms are developed that address these complexities. Evaluation of these algorithms indicates that for the configurations considered, the partitioned architecture is competitive, and warrants further investigation.

Finally, further insights into the novel architecture and dynamically-scheduled processors in general are developed through an analysis of the non-determinism introduced by dynamic scheduling, and the challenges this factor presents to computer architects who must design systems that can be used to full advantage by software. It is shown that static instruction scheduling for dynamically-scheduled processors is insolvable with known techniques.

Acknowledgements

This dissertation would not have been possible without the help of many people. I am indebted to my supervisors Dr. Paul Chow and Dr. Zvonko Vranesic for their advice, encouragement, and support while I developed the ideas set out in this dissertation. These ideas are the culmination of my work in many areas of computer architecture, and I thank Dr. Chow and Dr. Vranesic for giving me the freedom to explore these areas, and for the friendship that developed during our work together.

I am also indebted to Dr. Norman Jouppi for his invaluable suggestions, comments, and answers, contributions that have deepened my understanding of microprocessors and the tradeoffs in their design. Through Dr. Jouppi, I have had access to the computers of Digital Equipment Corporation's Western Research Lab. I thank Dr. Jouppi for this access, Annie Warren, Jason Wold, and Dr. Alan Eustace for logistical support while I ran my simulations, Digital Equipment Corporation for the use of these computers, and the members of the Western Research Lab for tolerating the simulations.

I wish to thank the members of my Ph.D. committee, Dr. Joel Emer, Dr. David Lewis, Dr. Todd Mowry, and Dr. Dave Wortman, for their careful reading of the dissertation and invaluable suggestions.

Many others have contributed to this dissertation through their experience, knowledge, enthusiasm, support, friendship, and distractions. Thank you Dr. Brad Calder, Qiang Chung, Mohammed Khalid, Fernando Nasser, Dr. Kathy Richardson, and Dr. Michiel van de Panne. Thanks also to Peter Pereira for his help with computer and network problems, and to Sarah Cherian, Gail Dragan, and Jennifer Rodrigues for their help with administrative matters. Finally, many thanks to the other (past and present) students, staff, and professors of the Computer and Electronic Groups who have made the past six years enjoyable.

I thank my family for their moral support. In particular, I would like to thank my mother for her encouragement and optimism, and my father for his careful reading of the final draft of this dissertation. Last but not least, I thank my wife Tanyl for her endless encouragement and patience. Your influence is present in this dissertation although you may not see it.

Financial support for this dissertation was provided by the Natural Sciences and Engineering Research Council (NSERC) of Canada. I greatly appreciate this support as otherwise I would not have had the opportunity nor freedom to pursue my interests in computer and non-computer issues. This work was also supported by an equipment grant from Digital Equipment Corporation.

Contents

List of Figures

Chapter 1

Introduction and Background

This dissertation presents an investigation of the memory-system design requirements in dynamically-scheduled superscalar microprocessors. This investigation is motivated by the recent advances in microprocessor architectures that offer significant performance potential but with significant hardware complexity. Such processors challenge computer architects to develop designs that balance hardware complexity and compiler technology against performance targets. The goal of this thesis is to develop an understanding of the hardware and software complexities that dynamically-scheduled processors introduce. The focus of the investigation is the register file whose design and resulting functionality helps define the functionality and the design of other system components. A secondary focus is the lockup-free data cache, a hardware-based prefetcher, and the interface between these components and the lower levels of the memory hierarchy.

This introductory chapter defines the framework within which the research is conducted and describes the basic goals of the research. The properties and functional requirements of dynamically-scheduled processors are examined in the next section. Then, the fundamentals of the dissertation are discussed, with Section 1.2.1 defining in more detail the goals of the research, Section 1.2.2 describing the methodology, and Section 1.2.3 indicating the major contributions. The chapter concludes with an overview of the remainder of the dissertation.

1.1 Fundamentals of Dynamic Scheduling

The continuing demand for higher performance is driving the development of superscalar processors that can execute ever larger numbers of instructions in parallel while maintaining the outward appearance of sequential execution. To exploit this parallelism, the machine-level instructions of an application must be ordered to uncover sufficient instruction-level parallelism. Traditionally, *instruction scheduling*, the process by which instructions are ordered, has been performed *statically* (i.e., at compile time) with the instruction schedule implicit in the generated program binary. This schedule is called the *static schedule*. When the binary is subsequently run on a superscalar processor, instructions are fetched from consecutive positions in the static schedule unless a branch or jump instruction redirects the fetch stream to a different part of the static schedule. The stream thus formed by the fetched instructions is then transformed by the hardware into a version that can be executed in parallel. After instructions have been issued to the functional units and their execution has been completed, the results are resequenced to give the appearance of being executed in the order in which they were fetched.

A *statically-scheduled* processor issues the instructions to the functional units in the order in which they were fetched. Such a processor will stall the instruction-issue process if the next instruction to be issued has an unresolved hardware or data dependence; in these situations, a *hazard* is said to exist. More recently, however, increasing numbers of processors are being introduced that can issue instructions out of fetch order. These processors, which are said to be *dynamically scheduled*, issue instructions out of order whenever a dependence-induced hazard prevents in-order issue; more precisely, the processor attempts to issue instructions from later in the fetch order that are not affected by the hazard. Thus, dynamically-scheduled processors are less affected by dependence-induced hazards that the compiler could not predict, or that it could not prevent by statically rearranging the instructions in the program binary.

1.1.1 Functional Requirements

A dynamically-scheduled processor comprises in essence three decoupled compute engines: the first engine fetches instructions from the memory system, the second engine executes the instructions by issuing them possibly out of fetch order to the functional units, and the third engine retires the instructions in order so as to implement the desired programming model.

Instructions are fetched from the memory system and are placed in a *dispatch buffer* from which the dynamic-scheduler selects those that will be issued next. This buffer is shown in Figure 1.1 and it is connected to the instruction memory via the instruction fetcher. Although all frameworks for implementing dynamically-scheduled processors have such a buffer, in some it is implicit in the design of the *reorder buffer* [Smith 1985], or of the *reservation stations* [Anderson 1967]. In others, it is explicitly implemented using a *dispatch queue* or *dispatch stack* [Dwyer 1992]; these implementation frameworks are described in Chapter 2.

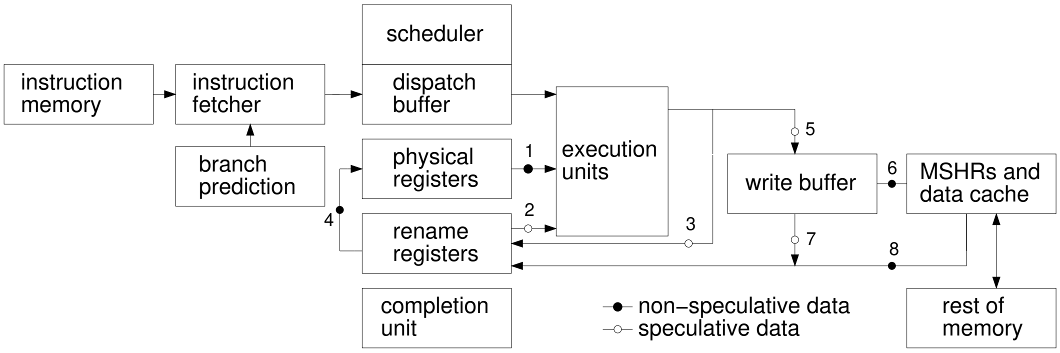


Figure 1.1: Block diagram of a generic dynamically-scheduled processor.

As instructions are inserted into the dispatch buffer, they are assigned a special hardware resource called a *rename register* and they are appended in order to an *active list*. This list keeps track of the fetch-order relationship between all active instructions, and is used for various purposes including the processing of exceptions. As in the case of the dispatch buffer, the list may be implicit in the design of the reorder buffer, or it may be explicitly implemented.

1.1.1.1 Branch Prediction and Instruction Orders

While the hardware limits the maximum number of instructions that can be issued in a cycle, the number actually issued is a function of: (1) the number of instructions with no unresolved dependences induced by instructions earlier in the fetch order, and (2) the availability of the required execution units. To increase the number of instructions that are actually issued, *branch prediction* and *speculative execution* can be used. Branch prediction permits the hardware to begin fetching the instructions following a branch before the branch is issued and its direction and destination are verified. Thus, when the hardware detects a branch in the instruction-fetch stream, it can begin fetching the instructions of the next basic block[1] without waiting until the outcome (i.e., direction and destination) of the branch is verified. Speculative execution permits the hardware to issue instructions before the outcome of all preceding branches have been verified. If the outcome of a branch is later found to have been incorrectly predicted, all instructions following the branch in fetch order are flushed from the processor.

To aid in the discussion of branch prediction, speculative execution, and dynamic scheduling, Figure 1.2 is presented. The purpose of this figure is to illustrate the five instruction orders for a dynamically-scheduled processor. Figure 1.2(a) shows a static schedule which comprises two basic

[1] A basic block is a sequence of instructions with transfers out of the block allowed from only the last instruction in the sequence, and transfers to the block allowed to only the first instruction.

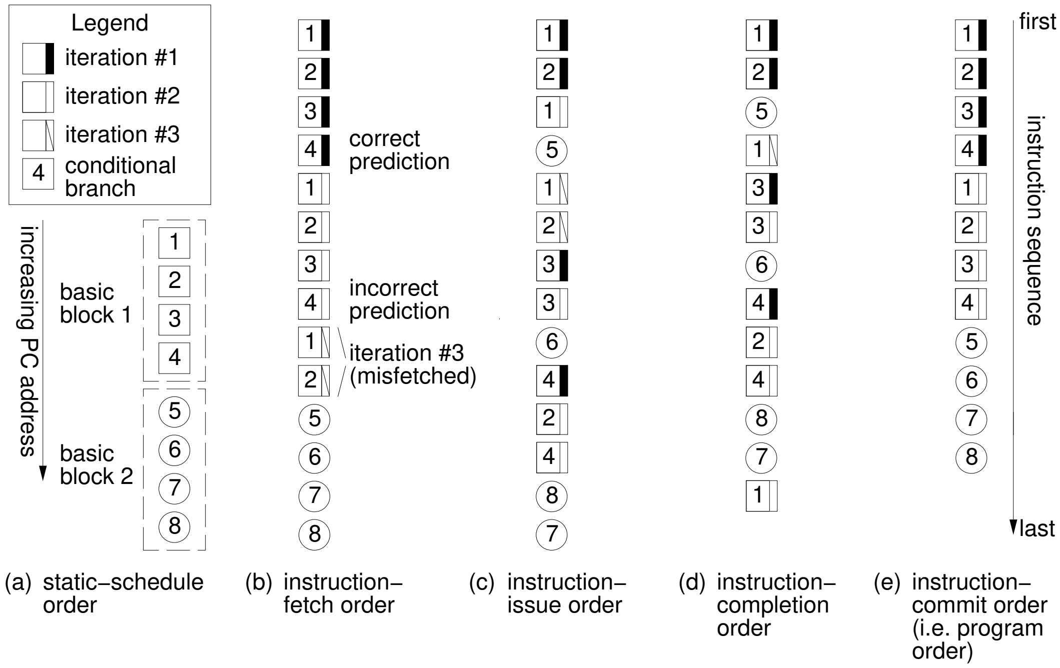


Figure 1.2: The instruction orders for a dynamically-scheduled processor. The fetch, issue, completion, and commit orders are execution dependent and were arbitrarily chosen for the purpose of illustrating the orders. These four orders assume that instruction #4 is a conditional branch, which is mispredicted at the end of the second iteration of basic block #1. Before this misprediction is detected, it is also assumed that the first two instructions of the basic block are fetched and issued, and that execution of the first of these is completed.

blocks and one conditional branch (instruction # 4), while Figures 1.2(b) to (e) show the fetch, issue, completion, and commit instruction orders that might occur at run time. These dynamic instruction sequences were arbitrarily chosen; the factors that might produce the illustrated orders are discussed below.

Assuming for the moment perfect branch prediction and no exceptions, it is guaranteed that if the first instruction of a basic block is fetched from memory, then all the remaining instructions in the basic block will also be fetched. However, if mispredicted branches or exceptions do occur, while the instruction stream may contain only some of the instructions from a particular basic block, the instructions that are present from this basic block will be in static schedule order. This scenario is illustrated in Figure 1.2(b). Figure 1.2(b) shows the instruction-fetch stream that might result from the execution of the static schedule if the conditional branch (instruction # 4) is mispredicted at the end of the second iteration of basic block #1. For this order to occur, the time delay between when the hardware predicts the outcome of the branch and when it verifies the prediction must be sufficiently long that two instructions are misfetched from the third iteration of basic block #1. As suggested by Figure 1.2(b), the fetch order is a function not only of the static schedule and the outcome of branches but also of the occurrence of mispredicted branches and exceptions. For this reason, statically- and dynamically-scheduled processors running the same program binary with the same data set may have different fetch orders.

Dynamic scheduling permits the instructions to be issued out of fetch order, and one possible issue order is shown in Figure 1.2(c). Note that the delay in issuing the second instance of instruction #4, the conditional branch, is assumed to be sufficiently long so as to permit the two misfetched instructions to be issued. An issued instruction is said to have been *completed* when the hardware can use the instruction's result to alter the state of the processor. A branch is said to have been completed when the branch outcome is verified, a store is said to have been completed when the data can be written into the cache or the write buffer, and any other instruction is said to have been completed when its result can be written into the destination register. Because functional units may have dissimilar latencies, instructions may not be completed in the order in which they were issued, as illustrated in Figure 1.2(d). Moreover, if the prediction for a branch instruction is discovered to be false, instructions following the branch in fetch order that have already been issued may not have completed, as is the case with misfetched instruction #2.

Once an instruction has been completed, the instructions following it in the fetch order can make use of the result or the side-effect it produced. An instruction is said to have been *committed* when the hardware has completed it *and* all the instructions preceding it in fetch order. There is typically a delay between the completion and commitment of an instruction. The order in which instructions commit is referred to as the *program order*. A committed instruction will never be reissued because all of the instructions before it in the program order have been completed. A completed instruction, however, will be issued again should the subsequent execution of any of the instructions preceding it in the fetch order give rise to an exception or a mispredicted branch. Figure 1.2(e) shows the commit order for the example; observe that the third instances of instructions #1 and #2 does not appear in the commit order. In general, in the absence of branch mispredictions and exceptions, the fetch and program orders would be the same.

1.1.1.2 Exception Models

Instruction speculation requires the ability to recover and resume execution after the detection of a mispredicted branch or a non-fatal exception. The steps taken in this process are dictated by the adopted exception model. Under the *precise exception* model, it is required that the instructions preceding the faulting instruction in fetch order be allowed to change the state of the system while those following the instruction will not. The latter instructions are flushed from the processor, and may later be re-inserted and re-executed. Thus, the state of the system will be restored to what it would have been had the faulting instruction and any that follow it not been executed. As discussed in Section 2.1.4, under this model, exceptions are recognized out of program order but they are processed in order. However, with mispredicted branches, the steps taken to recover from a mispredicted branch are usually begun as soon as the misprediction is detected. Under *imprecise exceptions*, the state of the system is not maintained so exactly; exception models are discussed in greater detail in Section 2.2.

1.1.1.3 Registers and Register Renaming

Once an instruction is completed and until it is committed, the result it produces is speculative because there is no guarantee that the instruction will not be flushed from the processor due to a mispredicted branch or exception. As a result, until instructions are committed, all changes to the state of the processor made by completed instructions must be reversible. To support reversal of these changes, temporary storage is provided to hold the speculative results of instructions. These results are made non-speculative and irreversible when the conditions of the employed exception model are satisfied. For simplicity, the following discussion assumes the use of precise exceptions. Under precise exceptions, the result produced by an instruction is made non-speculative and irreversible when the instruction is committed.

The storage for the speculative and non-speculative values of the instruction-set-architecture (ISA) registers is provided by the hardware registers. The ISA registers are the registers that can be named by the instructions, and these are referred to as the *architectural registers*. The hardware registers that are used to hold speculative values of the architectural registers are referred to as the *rename registers* while the hardware registers that are used to hold non-speculative values of the architectural registers are referred to as the *physical registers*. At any point in time, there may be many rename registers each holding a speculative value for the same architectural register R, but there will be at most one physical register holding the non-speculative value for R.

To illustrate the above concepts and the register-renaming process, consider the insertion of an instruction I into the dispatch buffer. When I is inserted, its source registers are mapped to the storage location that contains the corresponding latest values. This location may be a physical register if the latest value is non-speculative, or it may be a rename register, if the latest value is speculative. Then, if I names a destination architectural register, this register is mapped to a unique and free rename register. When the instruction is subsequently issued, the source registers are read, a process requiring one or both of paths #1 and #2 in Figure 1.1. Once the instruction is completed, its result is written to the rename register using path #3. And finally, when the instruction is committed, the speculative value is copied from the rename register to the appropriate physical register using path #4, and the rename register is freed. Depending on the implementation framework for the processor, the renamed registers may be explicit registers, or may be implicit in the design of the reservation stations or the reorder buffer, as discussed in Section 2.1. Also depending on the implementation framework, copying the contents of a rename register to a physical register may not be necessary; a processor that uses such a framework is discussed in Section 2.1.4. Nonetheless, the notion of a rename register is still valid and useful.

Renaming registers on insertion has two advantages: instruction-issue deadlock is avoided, and non-essential data hazards are eliminated. Issue deadlock will occur if storage is allocated when instructions are issued and the following two statements are true: (1) an instruction I requires a rename register, and because there are none free, it cannot be issued; (2) all the rename registers are currently allocated to (already-issued) instructions that are later in fetch order than I. This scenario leads to deadlock because a rename register can't be freed until these later instructions commit, but they cannot be committed until instruction I is committed, and instruction I can't be issued until a rename register can be allocated to it.

A second advantage of on-insertion renaming is the elimination of *write-after-write* and *write-after-read* hazards. A write-after-write hazard is said to occur if two instructions W_1 and W_2 both write the same register and W_2 is performed out of fetch order, that is, it writes the register before W_1 does. In this case, the register will have the wrong value after the writes are performed. A write-after-read hazard is said to occur if a read instruction R is located before W_2 but after W_1 in fetch order and is supposed to read the value V written by W_1, but before R can do so, W_2 overwrites V with the new value. Without on-insertion renaming, the instruction scheduler must prevent these hazards from occurring, with the simplest method being to respect all true-data and other dependences. In so doing, however, the amount of instruction-level parallelism is likely to be significantly reduced. With renaming, each write is assigned to a unique rename register, eliminating hazards induced by storage conflicts. Thus the instruction-level parallelism is constrained only by true-data dependences.

1.1.1.4 Loads and Stores

Buffering of store data is required both to allow the recovery from exceptions and mispredicted branches and to ensure that no write-after-write or write-after-read hazards occur. Ensuring that no such hazards occur means that a load instruction L must return the value written to the same memory address by a store instruction S provided that S occurs earlier than L in fetch order and no other stores exist to that memory location between S and L, in fetch order. Since memory renaming is not feasible due to the large size of memories, hardware is employed to dynamically detect and prevent the occurrence of these hazards; this hardware is called a *memory disambiguator*. After a store instruction is issued and after the memory address is calculated, the hardware checks if this address was read by a previously-issued load instruction that is *later* in fetch order than the store. If there is such a load instruction, it and the instructions that follow it in fetch order must be flushed from the processor and subsequently re-issued and re-executed.

When a store instruction is issued, the data item to be stored is not written into the data cache or other levels of the memory hierarchy until the store instruction is committed. Further, until the store is committed, one option is to buffer the data in a *non-merging write buffer* or a *write reorder buffer*. The data item is written into this buffer using path #5 in Figure 1.1. When a store is committed, path #6 is used to write the speculative data item to the data cache. When a load instruction is issued, the load address is compared against those of previously-issued store instructions that are *earlier* in fetch order than the load and that have not yet committed. If a match is detected, the load may be resolved using the uncommitted, that is, the speculative data; path #7 is used in this case. Alternatively, the load may be stalled until the store instruction is committed. If the data is not speculative, the load is resolved using path #8.

When store instructions are committed or loads are resolved using non-speculative data, the cache-block containing the data may need to be brought into the data cache. To allow other cache accesses to proceed during the resolution of a cache miss, a *lockup-free cache* [Kroft 1981] is used. This type of cache has special hardware buffers called *miss status holding registers* (MSHRs) [Kroft 1981] to hold information about outstanding misses so that these may be completed once the data is returned. Should a cache miss occur and there are insufficient miss-handling resources available, the processor must stop issuing memory instructions until a resource is freed. A cache miss that stalls the issuing of memory instructions is called a *structural-stall* miss because it induces a structural-hazard stall. Assuming an MSHR is available, it is allocated on the first miss to a cache block with a given tag; in this case, the miss is referred to as a *primary miss* [Kroft 1981]. This MSHR is used to record information for resolving this miss and for resolving any subsequent misses that occur to the same block while the block is being fetched. These subsequent misses are called *secondary misses* [Kroft 1981]; the hardware does not initiate a cache-block fetch for a secondary miss.

MSHR organizations for dynamically-scheduled processors must provide functionality that is not required in statically-scheduled processors. This functionality allows selective cache misses to be suppressed while allowing others to be completed. To illustrate the need for this suppression, consider the following in-fetch-order sequence of three instructions $load_1$, $branch$, $load_2$. Suppose that both loads access the same cache block, that they are both issued before the branch, and that the cache block is not in the cache. Now suppose that before the cache-block fetch completes, the branch is issued, and it is found to have been mispredicted. To recover from the branch misprediction, it is necessary to flush $load_2$ and those instructions following it in fetch order from the processor. When the cache block subsequently returns, while the rename register allocated to $load_1$ is written, the one that was allocated to $load_2$ must not be. Thus, a mechanism is required to delete the information about $load_2$ from the cache block's MSHR entry. Note that if all the misses for a given in-progress cache-block fetch are suppressed, the cache-block need not be written into the data cache, thereby preventing the possible ejection of required data. One method for implementing an MSHR organization is discussed in Section 2.1.4.

Dynamically-scheduled processors hide the cache-miss penalty by issuing instructions later in fetch order before the cache miss is resolved and before the instructions that are dependent on the cache miss are issued. This increase in instruction-issue disorder, however, may adversely impact performance as more disorder may lead to longer lifetimes for speculative values, to more branch mispredictions, and to more data-cache misses[2]. To lessen the disorder in instruction issue and its potential negative performance impact, the effective cache-miss latency can be reduced using prefetching. The goal of prefetching is to bring data closer to the processor before the processor requires it, rather than, as is the case with non-blocking loads, waiting for a cache miss to occur before initiating a fetch for missing data. Software-based and hardware-based prefetching are discussed further in Section 2.3.

1.2 Thesis Fundamentals

This section discusses the motivation for this dissertation, the goals of the research, the methodology used to address these goals, and the contributions of the research.

Dynamically-scheduled processors can achieve higher levels of performance than statically-scheduled processors for two reasons. First, they can exploit statically invisible instruction-level parallelism, and second, they can issue instructions out of fetch order whenever dependence-induced hazards prevent

[2] Data-cache misses may increase if there is an increase in the number of loads that are incorrectly speculatively executed, or if the increased disorder in instruction issue reduces the locality of data accesses and thus increases the conflict miss rate.

in-order issue. It is not possible to statically determine when out-of-order issue will occur because the hazards are a result of events that cannot be statically predicted. These events include data-cache misses, memory-location conflicts, and hardware-resource conflicts. Consequently, the order in which instructions are issued is not statically predictable. Furthermore, the state of the processor is also not deterministic since it is not possible to statically determine the instructions that will be concurrently in some stage of execution, and the stage each will be in. To ensure that instruction ordering does not affect the run-time behavior of an application, hardware is required.

The performance of a dynamically-scheduled processor is in part defined by the number of instructions that can be simultaneously in some stage of execution. Because hardware resources are required to track the status of all instructions, the number of these resources has a direct impact on performance. There must be sufficient resources to permit the software to take full advantage of the available hardware. Maximizing performance, however, is hindered by the run-time hazards that the additional hardware induces, hazards that cannot be covered by the dynamic scheduler or predicted at compile time. The inability to predict these hazards statically is due to the inability to statically predict resource availability, which is a by-product of the non-deterministic issue order. The number and type of hardware resources are limited by the cost of their implementation, and thus, a cost-effective design represents a balance between hardware complexity and compiler technology. It is therefore paradoxical that while dynamically-scheduled processors offer the potential for higher levels of performance than can be obtained with statically-scheduled ones, the non-determinism introduced by dynamic scheduling impedes static generation of code that is well tuned to the processor hardware. The performance impact of this paradox, however, is dependent on the design of a processor and its performance sensitivity to run-time hazards.

1.2.1 Research Goals

The primary goal of the research is to expose and investigate the hardware and software complexities that dynamically-scheduled processors introduce. This dissertation represents a first thorough look at these issues and establishes a framework that could be used to further investigate the cost-effectiveness of dynamically-scheduled processors. The primary goal is addressed by focusing on three specific areas.

- The main focus is the register file. The investigation of the register file attempts to quantify the tradeoffs in its design, and to isolate the design and performance relationships between it and other system components.

- A related focus is the other components of the memory system that are used to service data-cache misses. By investigating these components, the demands that the memory-system components place on each other are identified. In addition, a study is conducted of the effectiveness of hardware techniques to reduce the time cost of servicing cache misses and to increase the tolerance for cache misses, in the context of a dynamically-scheduled processor.

- Finally, the aforementioned software-hardware paradox is examined with the goal of creating a better understanding of the challenges facing computer architects who must design systems that can be used to full advantage by the software.

1.2.2 Research Methodology

The results in the dissertation are based on quantitative evaluation of a region in the design space of dynamically-scheduled processors. This section motivates this choice of methodology.

The register file was chosen as a focus for two reasons.

1. A wide-issue dynamically-scheduled processor is likely to have a number of components that are on the critical timing paths of the processor and whose designs could significantly impact performance. The register file, the dispatch buffer, and the register renaming unit are three such components. The implementation size and complexity of these components tend to scale together since it is desirable that no one structure offers a disproportionate amount of functionality. The register renaming unit and the dispatch queue are subject to wide variations in implementation architecture and circuitry, while the register file design space is more limited. Nonetheless, independent of how the dispatch buffer and renaming unit are implemented, they will have structures similar to those found in the register file (such as numbers of ports or numbers of entries).

2. Although many other researchers have investigated dynamically-scheduled processors that use register renaming, there is no known research that has focused specifically on issues affecting the register file.

The functional design of a register file can be described by the clock cycle count required to read or write a register, by the number of registers, and by the number of read and write ports. Its design is influenced by the other system components described in Section 1.1.1. These components place demands on the register file by virtue of all the components participating in the task of executing instructions and the instructions, in turn, sharing a limited number of components. Resource conflicts may therefore occur between instructions that require an instance of the same type of resource. Because these conflicts are resolved by the passage of time, to evaluate the design requirements of the register file and the memory system, it is necessary to include sufficient details in the model of the system to capture the resource requirements and the period of time over which each is in use.

The instruction-induced interaction between components gives rise to an unfortunate and unavoidable tradeoff between the amount of detail modeled and the generality of the evaluation. A very detailed model, for example, would permit an accurate estimation of the performance impact of a register file design for processors with the same parameters as those modeled. However, this estimate could not be extended to other processors with slightly different parameters. A less detailed model would give a less accurate prediction but would be representative of a wider class of processors.

Because time cannot be ignored, it is not possible to accurately investigate a processor by accurately modeling and investigating, in isolation, the components of the processor. Thus the model used for an investigation must balance the desire to accurately model certain features of a processor, and the desire to be representative of processors in general. Irrespective of the level of detail, the design space for dynamically-scheduled superscalar processors is very large owing to the many different techniques that have been used to implement them and the many different parameters that define these techniques. Thus, it is impossible to exhaustively explore the design space.

In light of the accuracy-generalization tradeoff and the size of the design space, the system models used in this research have been selected to be representative of a region in the design space. This modeling strategy has been chosen to permit the identification of performance trends and design relationships rather than to evaluate the performance of specific design points. These trends and relationships may be used to select a more limited number of design points at which further investigation is warranted.

The system model, which is described in Chapter 3, implements dynamic scheduling using a single dispatch queue (i.e., a dispatch buffer), an integer register file and a floating-point register file. Each register file contains the physical and rename registers and a mechanism for identifying the registers containing the visible state. Thus, no copying is required between rename and physical registers. This basis for the model was chosen owing to its simplicity and owing to the clean separation between dispatch-queue entries and rename registers. Nonetheless, since the investigation is performed at the functional level, the results of this investigation apply to implementation frameworks built around a reorder buffer or a centralized reservation station.

1.2.3 Research Contributions

This dissertation provides the following main contributions:

- A detailed architectural description, with practical implementation in mind, of important features of dynamically-scheduled processors that affect the design of the register file and of the memory system. The existence of such a description will help reduce the number of research projects whose results are not meaningful due to a failure to consider/control important parameters.

- An analysis of the non-determinism introduced by dynamic scheduling, and the challenges this phenomenon presents to computer architects who must design systems that can be used to full advantage by software.

- An investigation of the relationships that exist between components of dynamically-scheduled processors including the register file, and the data memory system.

- An analysis of how various design decisions impact the cycle time of a register file. The conclusions from this analysis motivated the investigation of a novel architecture in which decentralization of key components is employed to reduce the cycle time of the processor. A comprehensive examination is conducted of the software and hardware complexities decentralization introduces in a dynamically-scheduled processor.

1.2.4 Dissertation Organization

This dissertation is organized into seven chapters.

- Chapter 2 describes the implementation frameworks for dynamically-scheduled processors with a focus on the techniques used to provide the functionality described in Section 1.1.1. The goal of Chapter 2 is to give insight into the complex issues in the design of a dynamically-scheduled processor, and to motivate the system model that is used in the research.

- Chapter 3 then describes the system model and the methodology used to select the details that were included. Chapter 3 highlights the issues involved in designing a system model, and the tradeoffs that must be made between accuracy and generality.

- Chapter 4 investigates the register-file design requirements for a dynamically-scheduled processor. The goal of Chapter 4 is to provide insight into the complexity-performance relationship involving the register file and the following system aspects: the issue width of the processor and the number of functional units, the size of the dispatch queue, and the type of exceptions employed.

- Chapter 5 investigates the performance relationships involving the components in the levels of the memory hierarchy that are used to service data-cache misses. Chapter 5 seeks to provide insight into the effectiveness of hardware-based techniques for reducing the apparent time-cost of cache misses or increasing the tolerance for data-cache misses. The components that are considered are the register file, the lockup-free data cache, the hardware prefetcher, and the interface between these components and the lower levels of the memory hierarchy.

- Chapter 6 presents the first comprehensive description of a novel decentralized architecture for dynamically-scheduled processors, which is called the *multicluster architecture* [Jouppi]. The development of this description and the analysis of the multicluster concept were performed as part of this dissertation. The results of the analysis and its implications are also presented in Chapter 6. The goal of the analysis is to refine the initial proposal, to expose the complexities of the architecture, and to develop an understanding of them.

- Chapter 7 continues the investigation of the multicluster concept with a description and evaluation of static-instruction-scheduling algorithms that were developed for the architecture as part of this dissertation. Chapter 7 also presents an analysis of the role static instruction scheduling plays in determining the performance of dynamically-scheduled processors in general.

- Finally, Chapter 8 draws conclusions from the three main investigations and suggests areas for future work.

Chapter 2

Implementation Frameworks

This chapter describes the implementation frameworks for dynamically-scheduled processors with a focus on the techniques used to provide the functionality described in Section 1.1.1. The chapter also presents the results of an analysis of several important aspects of these frameworks, and enhancements to one of these frameworks that followed from the analysis; the analysis was performed as part of this dissertation. The goal of this chapter is to give insight into the complex issues in the design of a dynamically-scheduled processor, and to motivate the system model that is used in the research described in this dissertation.

The chapter begins with a description of the implementation frameworks as they are used in existing processors, and a presentation of an analysis of the common aspects of these processors. Then, an examination is presented in Section 2.2 of the exceptions models that are implemented by these processors. These two sections provide the foundation for the system model used in this dissertation. Finally, a discussion is presented in Section 2.3 of hardware-based prefetching, and in particular, of the proposed extensions to Jouppi's original stream-buffer model [1990]. This section also describes how the model was incorporated into the system model used in this dissertation. The system model is described in detail in Chapter 3.

2.1 Processor Implementations

All dynamically-scheduled processors speculatively issue and execute instructions out of fetch order and thus require some form of register renaming. This technique was pioneered in the design of the reservation stations for the IBM 360/91 [Anderson 1967]. Reservation stations provide the functionality of a dispatch buffer, which, as discussed in Section 1.1.1, maintains the pool of instructions from which the scheduler chooses those to issue next. Reservation stations alone, however, cannot be used to implement dynamic scheduling, because a mechanism is required to prevent speculatively-executed instructions from performing any updates until it is certain that the instructions will commit. One method for ensuring that updates can be undone is to use a reorder buffer [Smith 1985], which is essentially a FIFO queue that is used to force speculative results to commit in order.

Reorder buffers and reservation stations are used to implement dynamic scheduling in the PowerPC 604 processor [Song 1994], the PowerPC 620 processor [IBM 1994], the Intel Pentium Pro processor [Gwennap 1995], and the AMD K5 processor [Christie 1996]. The AMD K6 processor [Halfhill 1996] (previously the NexGen 686 processor), and the HP PA-8000 processor [Gwennap 1994b] also use a reorder buffer, but for these two processors the reorder buffer also provides the functionality of reservation stations. A different implementation framework, namely one based on dispatch queues and explicit register-renaming hardware, is used by the MIPS R10000 processor [Gwennap 1994a, Yeager 1996] and the DEC Alpha 21264 [Gwennap 1996] processor. Seven of these processors are examined in the following sections along with issues relating to lockup-free caches, exceptions, and branches.

2.1.1 The PowerPC 604 and 620 Processors

The PowerPC 604, as shown in Figure 2.1, has six reservation stations of two entries. Each reservation station stores the pool of instructions that can be issued to the functional units associated with the reservation station. A reservation station allows the dispatch unit to dispatch instructions before their source operands are available and before suitable functional units are available. An instruction is dispatched in fetch order to the reservation station of the target functional unit while, at the same time, an entry in the reorder buffer is assigned, and if the instruction names a destination register, a rename register is allocated. If any of these resources is not available, the instruction distribution is stalled until one becomes available. To simplify instruction distribution and recovery from mispredicted branches, stalls also occur if an attempt is made to dispatch a second instruction to a functional unit in the same clock cycle or if the previously-dispatched instruction was a branch.

In addition to allocating a rename register, called a *rename buffer* in the PowerPC nomenclature, the instruction is supplied with its source operands when it is dispatched to a reservation station. There are three cases to consider for each source operand:

1. The operand corresponds to a non-speculative value, in which case the value is forwarded to the reservation station from the appropriate register,

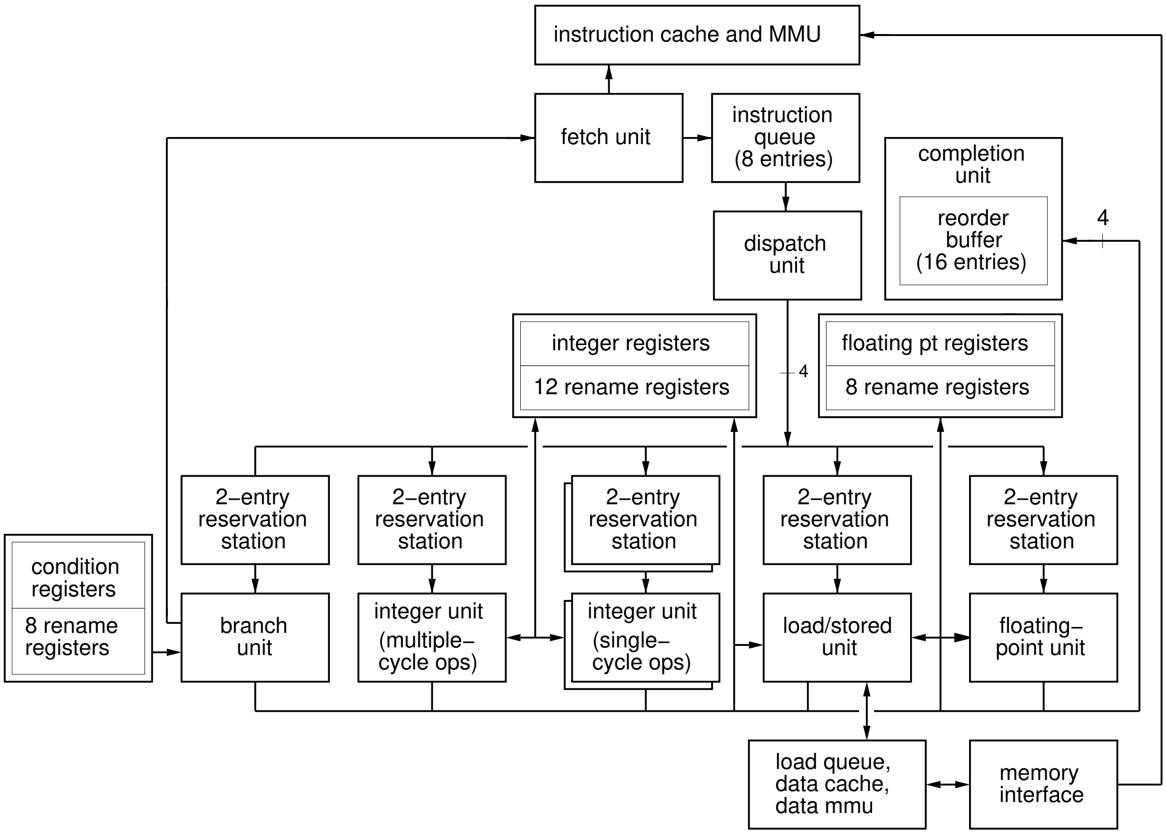


Figure 2.1: Functional block diagram of the PowerPC 604 processor.

2. The operand corresponds to a speculative value that is available, in which case it is forwarded to the reservation station from the appropriate rename register, and

3. The operand corresponds to a speculative value that is not yet available, in which case the reservation station is sent the tag of the appropriate rename register.

The tag supplied in the third case will be used by the reservation station to capture a copy of the value when it is available and being written to the designated rename register. When all the operands of an instruction are available, the instruction may be issued. Instructions may be issued out of order across the six execution units as well as within the integer execution units, which have out-of-order issue reservation stations.

After issuing a load instruction, the load/store unit first calculates the effective virtual (memory) address and the corresponding physical address, and then probes the data cache. If an exception occurs when calculating the physical address, the exception is noted for future processing; Section 2.1.6 describes the steps taken to process an exception. If the outcome of the data-cache probe indicates that the data is in the cache, the data is forwarded to both the designated rename register and the instructions in reservation stations that are waiting for the data. If the data is not in the cache, the load instruction is inserted into a four-entry load queue along with the physical address of the missing data, and it is held there until the cache block containing the data is fetched. When the data is returned, as with a hit, the data is forwarded to the required rename register and any waiting reservation stations. The mechanism used to implement this forwarding is discussed in Section 2.1.5 in the context of the R10000. The PowerPC 604 supports up to two in-flight cache misses but the first-detected miss must be resolved before the second can be. Thus, the processor offers a *hit-under-miss* capability.

For a store instruction, the cache is not probed when the store is issued. Rather, a store is considered completed after its physical address is calculated. When the store is committed, the data is read from the associated *physical register*, and written into the cache, with a cache-block being fetched if necessary. The physical register is read when the store is committed because the instruction that generated the value will have already been committed and its result will have already been copied to the appropriate physical register. While a cache block is fetched to service a store miss, subsequent stores in program order can be committed. Thus, loads and stores that hit in the cache can bypass loads and stores that are earlier in program order.

In each cycle, up to four instructions can be committed. If the instruction at the top of the buffer incurred an exception, it and the instructions following it in fetch order are flushed from the processor, and the exception handler is called. Otherwise, if no exception occurred during its execution, the instruction may be committed if all of the following are true:

1. It has been completed,

2. It is either not a store or it is the first instruction to be committed this cycle, and

3. The previously-committed instruction in this cycle was not a taken branch.

Conditions 2 and 3 are imposed to simplify the hardware. Condition 2 makes it unnecessary to provide hardware to allow a store to be committed in the same cycle as the instruction that generated the store data, while condition 3 makes it unnecessary to provide hardware for adjusting the program counter if a taken branch instruction is not the last instruction committed in a cycle. Once an instruction is committed, the top-pointer of the reorder buffer is moved to the next instruction in program order.

The microarchitecture of the PowerPC 620 processor is very similar to that of the PowerPC 604. The PowerPC 620, however, implements a 64-bit data path instead of a 32-bit data path, a more aggressive branch predictor, four fewer integer rename registers but eight more registers for the condition code register file, a load/store reservation station that can issue out of order, two additional reservation-station entries for branches thereby allowing speculation beyond four unresolved branches, a more aggressive translation lookaside buffer [Patterson 1995], and a larger data cache. The functional behavior of the PowerPC 620 is similar to the description presented above, with the exception of the complications induced by issuing loads and stores out of order. Similar complications arise in the PA-8000, and these are discussed in Section 2.1.3.

2.1.2 The Intel Pentium Pro, AMD K5, and AMD K6 Processors

The Intel Pentium Pro, AMD K5 and K6 processors differ from the other processors discussed in this chapter in that the instruction-set architecture is based on x86 instructions rather than RISC instructions. However, these three processors execute x86 instructions by converting them to RISC-like operations called ROPs by AMD, micro-operations or uops by Intel, and RISC86 instructions by NexGen. The micro-operations of the processors are so called because they have a fixed length and use consistent encodings. The AMD K6 processor is the more aggressive of the three with most of the additional complexity lying in the instruction decoding and dispatching logic. Both the AMD K5 and the AMD K6 predecode the x86 instructions before inserting them into the instruction cache. This predecoding reduces the complexity of the fetch logic because it need not handle variable length instructions. In addition, both of the AMD processors employ more flexible decoders to convert x86 instructions into the ROPs, thereby reducing the incidence of instruction stream stalls.

Like the PowerPC 604, the Pentium Pro processor uses a reorder buffer, as shown in Figure 2.2, but, unlike the PowerPC 604, it has a single reservation station with 20 entries rather than six separate reservation stations with a total of 16 entries. Instructions are fetched from the instruction cache, and converted into the micro-operations of the processor by three parallel decoders. These micro-operations are then dispatched in fetch order to the reservation station, from which they are then issued out of order to the functional units.

The Pentium Pro does not provide explicit register renaming hardware. Instead, a speculative result is held in the reorder-buffer entry assigned to the instruction until the instruction is committed. Thus, the rename registers are implicit in the design of the reorder buffer. The Pentium Pro therefore has 40 rename registers (one per reorder-buffer entry), while the PowerPC has 20 general purpose (rename) registers. When an instruction is committed, as in the case of the PowerPC 604, the hardware copies the speculative result into the physical registers (the *real register file*, in Intel nomenclature) from the appropriate reorder-buffer entry (i.e., rename register). Unlike the PowerPC 604, the Intel Pentium Pro allows stores to be speculatively executed with the processor using a memory reorder buffer to ensure

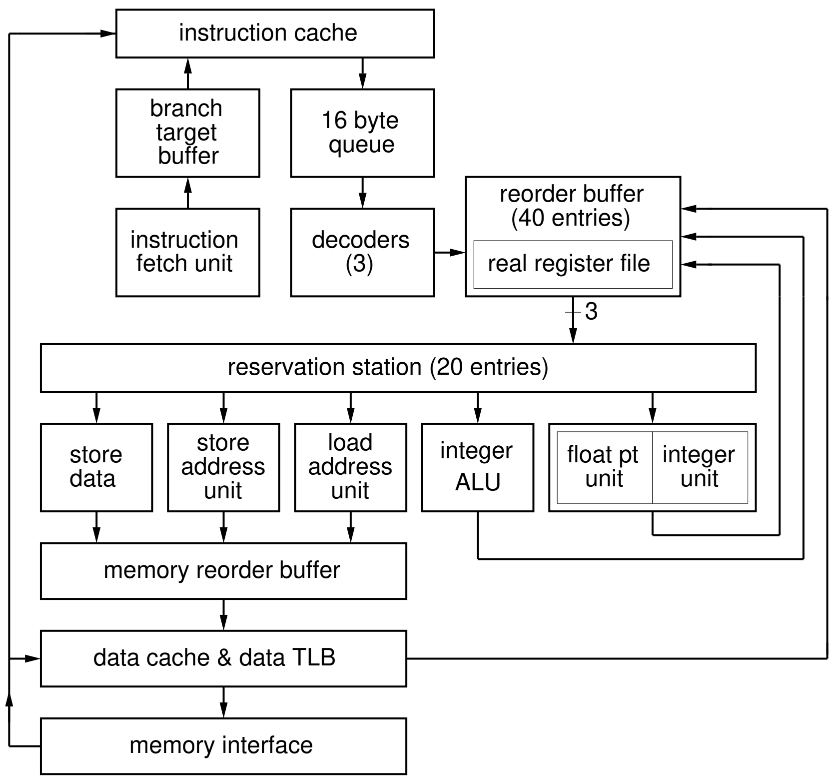


Figure 2.2: Functional block diagram of the Intel Pentium Pro processor.

memory locations are updated in the correct order. But, as in the case of the PowerPC 604, the Pentium Pro's cache supports hit-under-miss.

Dynamic scheduling in the Pentium Pro is provided by a 20-entry reservation station, and a 40-entry reorder buffer for sequencing instructions and holding speculative values. The AMD K5 implements dynamic scheduling using a 16-entry reorder buffer and 8 reservation stations, each associated with a separate functional unit as in the case of the PowerPC 604. The AMD K6 has a 24-entry reorder buffer for both sequencing instructions and for holding the instructions until they can be issued. Therefore, this buffer implements the functionality of the reorder buffer and the single reservation station of the Pentium Pro, but it does not hold speculative results. Rather, a separate register file of 40 rename registers is provided, thus supplying the same amount of speculative storage as the Pentium Pro.

2.1.3 The HP PA-8000 Processor

The HP PA-8000 processor (Figure 2.3) like the AMD K6 processor, is based on a reorder buffer and rename registers. The PA-8000 processor is a RISC processor with considerably more functionality and thus performance potential than the PowerPC 604, the Pentium Pro, or the AMD K5 and K6 processors.

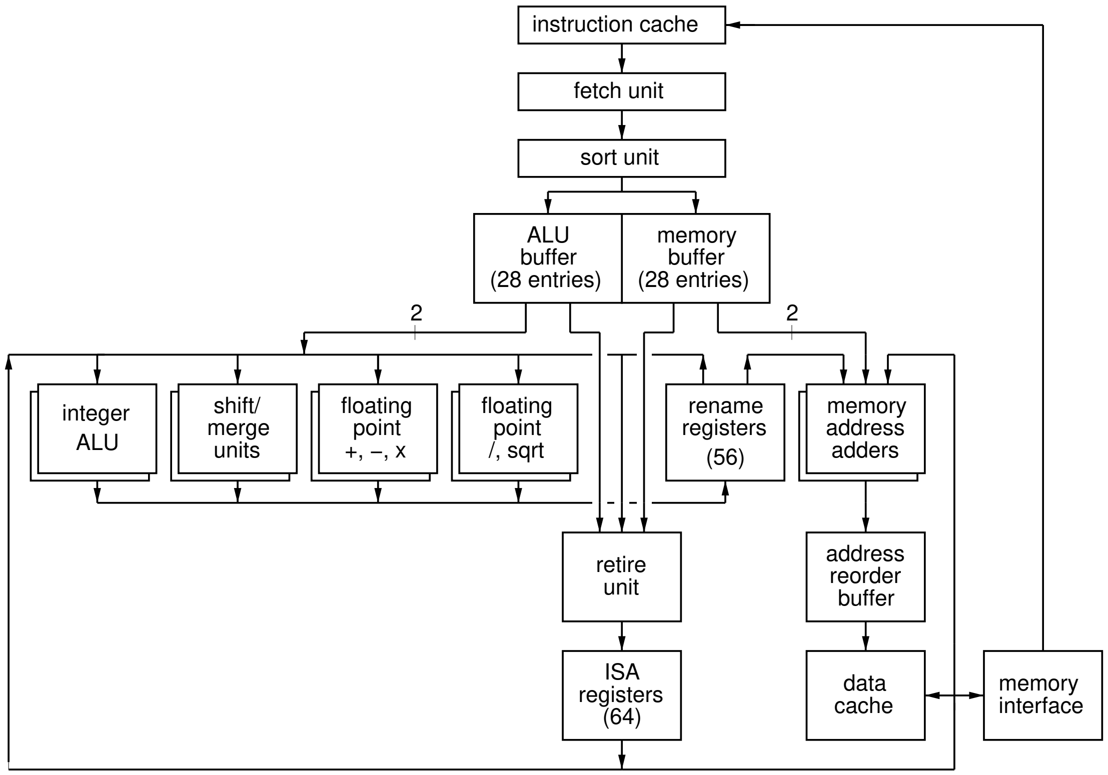


Figure 2.3: Functional block diagram of the HP PA-8000.

The 56-entry reorder buffer is split into two logical buffers, which was probably done to reduce hardware complexity. Both reorder-buffer halves hold the designated instructions from the time they are fetched until the time they are committed, with the 28-entry memory buffer holding only loads and stores, and the 28-entry ALU buffer holding all other instructions. For each reorder buffer entry, there is a rename register, and thus there are never any stalls in the insertion of instructions caused by a lack of rename registers. Each reorder buffer can issue up to two instructions per cycle; instructions are issued after all their input dependences have been resolved. To reduce the complexity of arbitration for the functional units, all functional units are duplicated, and each unit is associated with the odd or the even slots in the appropriate reorder buffer. Thus, for example, the instructions in the even-numbered slots of the ALU buffer compete for integer ALU0 while those in the odd-numbered slots compete for integer ALU1. Arbitration for ALU0, say, is won by the instruction in an even-numbered slot that is earliest in fetch order. Up to four instructions can be committed in each cycle, at which point the corresponding reorder buffer entries are freed, and speculative values are copied from the rename registers to the corresponding physical (i.e., ISA) registers.

Memory instructions arbitrate for the memory address adders in the same way as other instructions arbitrate for the other execution units. The winning instructions are dispatched to the memory address adders to have their effective address calculated. The effective addresses are then written into one of the 28 entries in the address reorder buffer (ARB). There is a one-to-one mapping between the ARB entries and the entries in the memory reorder buffer. Once the physical address has been calculated and recorded in the corresponding ARB entry, the ARB performs memory disambiguation. If the instruction is a store, the ARB determines whether the issuing of this store will induce a write-after-read hazard with a previously issued load that occurs after the store in fetch order. If so, all matching load instructions and those instructions that follow them in fetch order are flushed from the processor. When a store is committed, as in the case of the PowerPC 604, the data is copied from the physical register where it now resides, and is written into the cache. Memory disambiguation for a load is slightly different. In this case, the ARB determines whether speculative or non-speculative data should be used to resolve the load by comparing the addresses of the uncommitted stores presently in the ARB that occur before the load in fetch order. If a store is pending, in other words if the data is speculative, the load must wait until the data has been written to the cache because there is no mechanism for forwarding speculative store data to a load. If a store is not pending (the data is not speculative) the load arbitrates for the cache.

As in the case of the address adders, the earliest instructions in fetch order are given higher priority for accessing the cache. Should arbitration be won by a load, a cache probe is done, and if the data is in the cache, it is loaded into the named rename register. If a cache miss occurs, the data is fetched from the next level in the memory system. The PA-8000 permits up to 10 in-flight cache misses using hardware that is likely similar to that used by the R10000 (see Section 2.1.5). Should the winning instruction be a store, a cache probe is done to determine if the cache block containing the store address is in the cache, but the data is not written until the store is committed. If the data is not present, then a fetch request for the missing cache block is launched.

2.1.4 The MIPS R10000 Processor

Unlike the six previously-described processors, the MIPS R10000 [Yeager 1996] does not use a reorder buffer but rather uses *dispatch queues* with an *active list* for instruction sequencing. A dispatch queue, like the reorder buffers and reservation stations used by the other processors, holds the instructions that are waiting to be issued to the execution units. Also as in the case of the reorder buffer used in the PowerPC 604, PowerPC 620, K6 and PA-8000 processor, the dispatch queue does not hold speculative results; this function is performed by the rename registers. However, unlike all the other processors, the R10000 does not differentiate between a rename register and a physical register. In other words, the processor does not contain separate storage for speculative and non-speculative values. Thus, there is

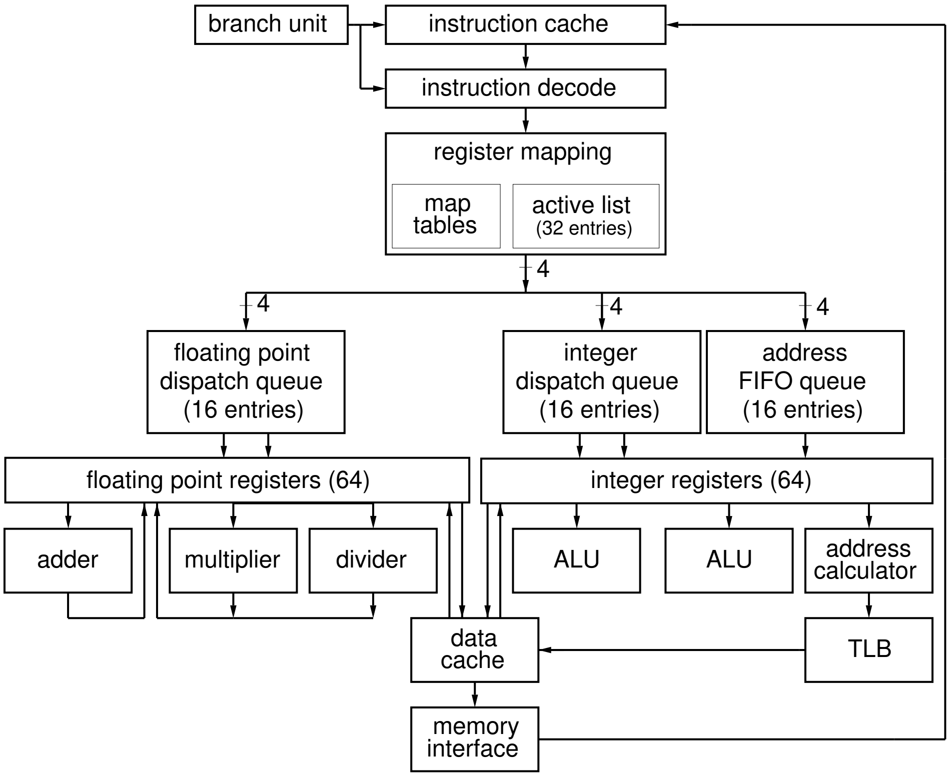


Figure 2.4: Functional block diagram of the MIPS R10000.

no need to copy the result of an instruction from one storage location to another when the instruction is committed. Instead, on committing an instruction, the processor changes the designation of the hardware register containing the now non-speculative value. For notational convenience, the hardware registers are collectively referred to as the physical registers.

As shown in Figure 2.4, the R10000 has three dispatch queues, one for floating-point instructions, one for integer instructions, and one for load and store instructions (the address FIFO). The three queues maintain for their respective execution units a pool of instructions from which the scheduling logic chooses the next instructions to issue. Instructions are fetched in sequence from the instruction cache and are inserted into one of three queues, with the exception of unconditional jumps with absolute addresses. These jump instructions do not need to be processed further and the encoded address can be used immediately to change the program counter and thereby redirect the instruction-fetch stream. As the instructions are inserted into one of the dispatch queues, they are added to the active list of instructions; this list keeps track of the fetch order of in-flight instructions. If among the 32 entries in this list there are no free entries, the instruction stream is stalled.

Also at insertion, the architectural registers named by an instruction I are renamed to physical registers. If I names an architectural register R_a as a source operand, the processor uses the *map* *tables* of the *register mapping unit* to determine the physical register R_p to which R_a was most recently mapped. In this way, I is provided with the physical registers containing its source operands. If I names an architectural register R_a as a destination, R_a is mapped to a free physical register R_p, and the appropriate mapping table (integer or floating point) is updated to indicate that R_a is currently mapped to R_p. When I is committed, the physical register currently containing the non-speculative value of R_a is freed, and the register R_p takes its place; this process is examined in more detail in Section 2.2.

The unification of the physical and rename registers simplifies the tracking of inter-instruction dependences and the acquiring of operands (operands need be read from only one place). Note that there will always be a physical register available because the processor supports no more than 32 in-flight instructions and there are 64 integer and 64 floating-point physical registers. In the worst case, if all instructions require physical floating-point registers, for example, there will still be 32 physical registers left over, which is a sufficient number to hold the non-speculative values of the 32 (architectural) floating-point registers.

Instructions are issued out of order to the functional units when their input dependences are resolved. For the integer and floating-point queues, if there are more instructions ready to be issued than can be issued in a cycle, instead of issuing the earliest in fetch order first, the circuitry prioritizes those ready to be issued by their position in the queue. The integer and floating-point dispatch queues do not track the fetch ordering of the contained instructions, probably so as to reduce the complexity of the queues and hence their cycle time. The discussion by Yeager [1996] suggests that the queues are on a critical timing path of the processor. By disregarding fetch order, instructions can be inserted into any free entry, while if fetch order is taken into account, before any new instructions were inserted, the queue would have to be compressed to free space at the end [Johnson 1991].

When an instruction is issued out of the integer or floating-point dispatch queue, its entry is released and can be reused in a subsequent cycle. Instructions that use the result of a load instruction are issued tentatively on the assumption that the load data is in the cache. However, if a dependent instruction is issued before the result of the cache probe is known and if the probe results in a miss, the dependent instruction is aborted, and is reissued in a subsequent cycle when the data is available. When a load or a store instruction is issued out of the address queue, the associated entry is not released, unlike in the case of the integer and floating-point queues. Instead, the entry is released when the instruction is committed. Since the address queue is essentially a FIFO, it is possible with less cost to rank the instructions by position in the fetch order, and thereby first issue those that are the oldest.

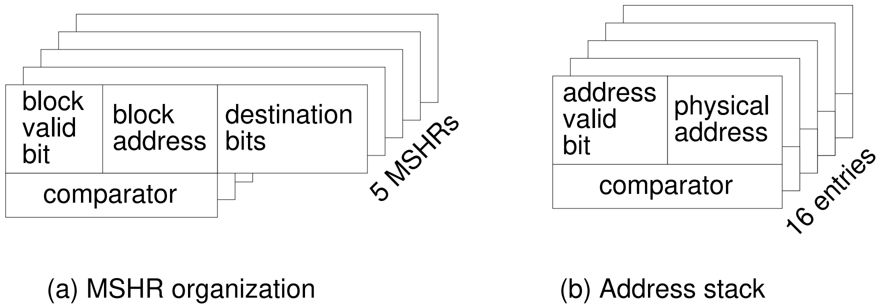


Figure 2.5: Functional block diagram of the proposed lockup-free cache implementation for the MIPS R10000.

2.1.5 Lockup-free Caches

This section describes a scheme for providing the R10000 with the capability to support multiple in-flight cache misses. This scheme was developed as part of this dissertation through analysis of the requirements of dynamically-scheduled processors. The goals of the description presented here are to provide the reader with further insight into the complexities introduced by dynamic scheduling, and to document an implementation detail of dynamically-scheduled processors that is not described in the literature.

When a cache miss occurs, the hardware must determine whether the cache-block containing the missing data is already being fetched, and must resolve all pending cache misses when this cache-block is returned from the lower levels of the memory hierarchy. To keep track of the in-flight cache-block fetches, a set of miss status holding registers (MSHRs) [Kroft 1981] is required. The R10000 requires four MSHRs because it can have four in-flight data-cache misses. However, a fifth MSHR is likely provided to permit the servicing of an instruction-cache miss to overlap the servicing of data-cache misses.

Figure 2.5(a) shows a block diagram of the MSHRs. In one of these MSHRs, the “block valid bit” is used to indicate whether the cache block with the address stored in the “block address” field is in the process of being fetched. At the same time, the “destination bits” indicate whether the cache block is to be returned to the data cache, the instruction cache, or perhaps neither because it is to be used to resolve an access to an uncached memory location. When a cache miss is detected, the MSHRs are searched associatively to determine whether the cache block is already being fetched. If it is not, a free MSHR is allocated, the block address is written, the “block valid bit” set, and a fetch request for the cache block is issued to the next level in the memory system. If there are no free MSHRs, no further memory requests can be issued until an MSHR is released by the return of previously-requested data. For a cache access, the load or store that caused this structural-hazard-induced stall (see page 8), will be reissued by the address queue subsequent to the freeing of an MSHR.

When a cache-block is returned, an associative lookup is again done of the MSHR structure to extract the destination bits. Note that some information must be returned with the data to permit this lookup to occur. Since there are so few MSHR entries, the information could be the MSHR number that was sent along with the fetch request. More probably, since the R10000 supports cache consistency and thus has a mechanism for sending addresses to the processor from the memory system, the fetch-request address could be returned using the same mechanism. After the destination bits are extracted, the cache-block is sent to the component that requested a copy.

For data-cache misses, an address stack is likely used to complete load or store instructions that missed in the cache. A possible block diagram for the address stack is shown in Figure 2.5(b). This stack might be implemented as a fully-associative buffer with 16 entries, each of which corresponds to exactly one entry in the address queue. An organization of this type would mirror the PA-8000's 28 entry memory buffer and associated 28 entry address reorder buffer. After the hardware issues a load or a store and calculates the physical address for the memory access, it writes this address into the address-stack entry that is associated with the address-queue entry of the instruction. Then, should the required data not be found in the cache, the valid bit in the address stack for the instruction is set, and if necessary, a cache-block fetch is initiated. When this block is returned from the lower levels of the memory system, at same time as the hardware writes it into the cache, it does an associative lookup of all the entries in the address stack. For each match, the control logic of the address queue notes that the instruction in the corresponding address-queue entry can now be reissued.

Note that it is possible to use the address stack in the same way as the address reorder buffer in the PA-8000 to detect inter-instruction memory dependences. To extend the proposed address stack to processors in which the entries are freed when their corresponding memory instructions are completed rather than committed, each entry may require an additional field and an associated comparator. These components are used to record and identify the memory instruction to which an entry corresponds. They are not required for the R10000 or the PA-8000 because, in the case of these processors, there is a permanent one-to-one mapping between the address-stack entries and the entries in the buffer that holds the memory instructions while they are in the system.

In the event that a mispredicted branch or an exception mandates the flushing of a memory instruction M from the processor as part of the flushing process, the address-stack and address-queue entries held by M are invalidated. Thus, if M was waiting for a cache block to be fetched, when the block is subsequently returned, it is guaranteed that the action indicated by M will not be performed, because there is no matching entry in the address stack. If M is a load, the named physical register will not be written, whereas if M is a store, the data will not be written. In this way, the memory instructions flushed by the branch misprediction or exception are suppressed while those preceding the faulting instruction can be completed normally (selective suppressing of cache accesses is discussed on page 8). This ability to easily suppress selective cache misses makes an address stack more attractive for dynamically-scheduled processors than the more conventional methods for recording information about primary and secondary misses. Farkas and Jouppi [1994] present a description of these methods. Note that should no matches occur during a probe of the address stack, the cache-block need not be written into the data cache.

Although not described in the literature, it is likely that lockup-free caches are implemented in a similar way in the PA-8000 using the memory buffer and the address-reorder buffer, in the PowerPC 604 and 620 using the load queue, and possibly in the other described dynamically-scheduled processors.

2.1.6 Exceptions and Mispredicted Branches

In addition to support for multiple in-flight cache misses, the processors described in this chapter require support for recovering from exceptions or mispredicted branches. The goal of this section is to provide the reader with insight into the implementation details of this support. Again, the description is presented in the context of the R10000.

If, during the execution of an instruction, an exception occurs, the exception is recorded but, as with the other processors examined in this chapter, it is not serviced until the hardware attempts to commit the instruction. In this way, exceptions are serviced in program order, and the exception handler is provided with a consistent in-order state when called. Moreover, in-order servicing of exceptions provides a simple mechanism for determining the program counter address of the instruction that raised an exception; the address is merely the address of the preceding instruction.

When an exception occurs, the R10000 notes the exception in the active list entry of the instruction, and the instruction is aborted. When the instruction is next in line to be committed, the register file is backed up and the instruction and those following it in program order are flushed from the processor. The register file is backed up by traversing the active list, starting with the most recent instruction and ending with the one that caused the exception. As each instruction is visited in this traversal, the information contained in its active-list entry is used to restore the register mapping table and register free list. As the R10000 can back up four instructions per cycle, backing up the register file can require up to eight cycles should all 32 instructions be affected by an exception. In addition to backing up the register file, all state information for any affected instructions must be invalidated. Because the R10000 processes exceptions in program order, every active instruction will be affected by an exception, and thus all instructions can be flushed at once. Therefore, in-order processing of exceptions not only simplifies the hardware required to flush instructions and resume execution, but also it gives the appearance that the processor implements a sequential execution model. A sequential-execution model precludes the overlap of the execution of any instruction by the execution of another. Therefore, under this model, the fetching of an instruction cannot begin until the previously fetched instruction has committed.

While it is acceptable that it might take up to eight cycles to recover from an exception, a faster mechanism is used for mispredicted branches since they are expected to occur more frequently [Johnson 1991]. A common technique is to record, for each predicted branch, its program counter address and some of the processor state before the branch is inserted into the processor. The R10000, for instance, has four special shadow registers, each of which is used to record the mapping table, the program counter and other information just before the branch passes through the register-renaming stage. Similar information is stored in the PowerPC 620 and the PowerPC 604, for instance, with the 620 supporting four in-flight branches and the 604 supporting two. If there is no available shadow register for a conditional branch, the instruction stream must stall until a register becomes available. Mispredicted branches are processed as soon as the branch is issued and the branch-condition is checked. When a mispredicted branch is detected, the stored information permits the register file to be backed up quickly, a process that the R10000 can do in one cycle. In addition, instructions that follow this branch in fetch order must be flushed from the processor, a process that requires selective flushing, and one from which rapid recovery is desirable. Hence, a more complicated mechanism is required than the "global flush" mechanism used for exceptions.

As each instruction is inserted into the processor, information is appended to it that indicates which of the pending branches have yet to have their predictions verified; a branch prediction is verified after the branch instruction is issued. This information is represented as a bit mask with each bit corresponding to only one branch. Thus, for the R10000 and the PowerPC 620, four bits are required, while the 604 requires only two. When an instruction I is inserted into one of the dispatch queues of the R10000, the bits of the mask are set to indicate which of the preceding four branches has yet to be verified; if the branches corresponding to bit positions 1 and 2 are in this state, then bits 1 and 2 are set.

After a conditional branch is issued, the prediction made for it is verified. If the prediction was correct, then a bit mask is broadcast to all buffer entries and functional units to cause the clearing of the bit corresponding to this branch. However, if the prediction was wrong, a bit mask is again broadcast with the bit corresponding to the branch set. This time, however, if there is a non-zero intersection between this mask and that of the instruction held in a functional unit or buffer entry, the instruction is invalidated. Thus, with one broadcast, all instructions following the branch can be flushed from the system. The broadcast that is performed when a branch is *correctly* predicted is required to permit the shadow register and associated bit to be reused before all control dependent instructions are committed. That is, assume there are two branches B_1 and B_2, with B_2 appearing later in fetch order than B_1. If B_1 is committed before B_2 is fetched, it is possible that both will be allocated the same shadow register. If this scenario occurred, the instructions that were control dependent on B_1 must be distinguishable from those that are control dependent on B_2. They must be distinguishable so that if B_2 is mispredicted, only the instructions following it in fetch order would be affected. By reseting the bit of a conditional branch once the branch outcome is verified, it is possible to distinguish between these two sets of instructions.

2.1.7 Implementation Complexity

In the preceding subsections, the microarchitectures of seven existing dynamically-scheduled processors were described. In this section, the results and implications of an analysis into the functional similarities of these architectures is presented. This analysis was performed as part of this dissertation.

Although the descriptions of the seven microarchitectures covered the details of interest to the research presented in this dissertation, many details were omitted for brevity; the reader is referred to the cited literature for more complete descriptions. If the implementation details of these processors are ignored, the techniques used to support dynamic scheduling are not all that dissimilar. In particular, all techniques provide the following functionality:

1. Tracking of fetch order,
2. Buffering of instructions waiting to be issued in some form of a *dispatch buffer*, and
3. Storage for speculative and non-speculative values and a mechanism to distinguish the two.

For tracking fetch order, although the R10000 uses an active list, and the other processors use a reorder buffer, from the functional perspective these two approaches are essentially the same. From the implementation perspective, however, these approaches do not have the same costs. The complexity and the design of the active list are likely to be similar to that of the reorder buffer in the PowerPC 604 and PowerPC 620 processors, because these processors use the reorder buffer in the same way that the

R10000 uses its active list. That is, the reorder buffer tracks fetch order and records information about the status of instructions, but it does not provide storage for the rename registers nor does it function as a dispatch buffer. For the other four processors, the reorder buffers used in the Pentium Pro and the K5 processors implement the rename registers, while the reorder buffers used in the K6 and PA-8000 processors implement the dispatch buffer.

There are two frameworks used for implementing the dispatch buffer. The PowerPC 604, PowerPC 620 and AMD K5 processors implement a decentralized structure using reservation stations, while the PA-8000 and R10000 processors implement a partially-centralized structure. On the other hand, the AMD K6 and Pentium Pro processors implement a completely centralized structure. The degree of centralization has an impact on the ability to fully utilize the available issue bandwidth of the processor. A characteristic of decentralized designs is the use of a separate buffer associated with each functional unit. However, since the buffers can issue instructions to only one unit, they will likely be underutilized. Underutilization occurs because instructions are distributed in fetch order to the buffers, thus mandating a stall if there are no free entries in the buffer for a particular unit. More centralized designs have fewer buffers but since each can issue instructions to more than one type of unit, it is less likely that the entries will be underutilized.

More centralized designs have more complex hardware in part because the issue and entry-management logic is more complex. With less centralized designs, although this logic must be duplicated, it will be less complex. Complexity concerns likely motivated the three-way partitioning used in the R10000 and the two-way partitioning used in the PA-8000 processors. With a single buffer (i.e., reservation station) per functional unit, each station issues only one instruction per cycle to one functional unit, thereby avoiding the complexity of multiported buffers and simultaneous deallocation of several entries. In addition, since the dynamic scheduler does not have to resolve functional unit conflicts or allocate buses to multiple instructions, and since it has fewer instructions from which to select one for issue, the instruction scheduling logic is simpler. Finally, a significant saving in complexity is possible by mandating that instructions issue in order within a reservation station [Johnson 1991], as is done in the floating point units of the PowerPC 604 and 620 processors. With this mandate, the reservation station can be implemented as a FIFO queue with only the head entry being inspected in any one cycle.

Finally, regarding the implementation of rename and physical registers, the processors that have been discussed in this chapter, except for the R10000, make a clear distinction between the storage provided for rename registers and that provided for physical registers. For the processors in which this distinction is made, to obtain the operands an associative lookup of the rename file must first be done to determine if the latest value of the operand is speculative. If it is, either its value or the rename-register number of the instruction that will produce the value is returned. If this lookup fails, then the value is read from the physical file. Because each file may contain the value, two sets of buses are required along with logic to prioritize the matches in the rename file and to select the latest value. Furthermore, since at some point the rename values might be copied to a physical register (i.e., to non-speculative storage), a write bus and support logic are also needed.

The R10000, however, does not require this hardware support because it does not use different storage for speculative and non-speculative values. Rather, both speculative and non-speculative values are held in physical registers, of which there are more than the number needed to maintain the architectural state of the processor. As an instruction is inserted into the dispatch queues, the ID of the physical register containing each operand is determined by a table lookup. Because this table is indexed by the architectural register number, no associative lookup is required. A similar table could also be used to avoid the associative lookup in implementations that have distinct rename and physical registers, such as is done in the Pentium Pro. When a speculative value is committed, the physical register containing the old value is merely retired, a process that does not require the copying of information. Thus, although the R10000 requires complex register mapping logic, the implementation seems simpler and less involved. Nonetheless, from the functional point of view, if the copy of rename registers to physical registers is ignored, both techniques are essentially the same.

Owing to the functional similarities between the approaches for implementing dynamic scheduling, and the clear distinction the R10000 approach makes between registers and the other system components, a processor model similar to the R10000 is used for the research that is discussed in the remainder of this dissertation.

2.2 Precise and Imprecise Exceptions

This section presents a more formal examination of the register renaming process and in particular of the conditions under which the speculative values contained in rename registers can be committed to the physical registers, followed by the reuse of the rename registers. This examination was conducted in the course of designing the system model used for the research discussed in this dissertation.

The examination is presented in this section in terms of processors like the MIPS R10000 that do not differentiate between rename and physical registers. However, in spite of this focus, which is adopted for clarity, the examination applies equally well to any dynamically-scheduled processor that employs register renaming.

In dynamically-scheduled processors, the physical registers can be freed only when their being freed will not prevent the processor from recovering and resuming execution after a mispredicted branch or non-fatal exception. The conditions under which a register can be freed depend on whether precise or imprecise exceptions are supported. The following concepts are key to the freeing of registers: the *completion* and *commitment* of an instruction, and the *creation*, *retiring*, and *killing* of an architectural-to-physical mapping. As noted on page 5, an instruction is said to *complete* when it reaches the point of altering the state of the machine, while an instruction is said to *commit* when it has completed *and* all the instructions preceding it in fetch order have completed. A committed instruction will never be reissued because all of the instructions before it in the fetch order have completed. However, a completed instruction will be issued again should the subsequent execution of any of the instructions preceding it in fetch order give rise to an exception or a mispredicted branch.

When an instruction I that names a destination register, say register R_v, is inserted in the dispatch queue, this register is renamed to a physical register, say R_p^1. When this renaming occurs, an architectural-to-physical mapping is said to have been *created*. As subsequent instructions in the fetch order are inserted into the queue, any that use register R_v as an operand will have this register renamed to R_p^1. This mapping between R_v and R_p^1 remains active until a subsequent instruction is inserted into the queue that names R_v as a destination. At this point, another physical register R_p^2 is mapped to R_v, and the $R_v \rightarrow R_p^1$ mapping is said to have been *retired*. A retired mapping is eventually *killed* and the point at which this killing occurs depends on the exception model. The physical register whose mapping has been killed is free for reuse.

To facilitate the discussion of the conditions for freeing registers, consider the scenario in which an architectural register R_a is named by an instruction I_1 as its destination register, and R_a has been mapped to a physical register R_p. Under precise exceptions, the register R_p will be freed when an instruction I_2 is committed if instruction I_2 is the first instruction after instruction I_1 in program order to have register R_a as a destination. The following requirements are inherent in this condition:

1. Instruction I_1 has been committed, and

2. The instructions that use the register R_p have been committed (these instructions occur later in program order than instruction I_1 and earlier than instruction I_2).

The condition for freeing registers ensures that the exact state of the machine can be recovered at any point should an instruction suffer an exception or a branch is mispredicted[1]. When an exception does occur or a branch is mispredicted, the mappings for each architectural register must be set back to the mapping that existed *before* the execution of the instruction causing the exception and any instructions following this instruction in fetch order. Conceptually, the resetting of the mappings entails moving a pointer in the architectural-to-physical register-map table, such as is done in the MIPS R10000 on a mispredicted branch. A similar table may exist in processors that differentiate between physical and rename registers, but in this case, it provides the architectural-to-rename register map; the Pentium Pro has such a table which is called the *register alias table*.

When the pointer is reset, the physical registers will still contain the correct values because a register R_p cannot be freed until all the instructions preceding the instruction writing R_p have been committed. That is, until all the instructions before an instruction I have been committed, the operands required by I will remain in the register file. A second step in the recovery from an exception or a mispredicted branch is that all the instructions later in the fetch order than the instruction that caused the exception are flushed from the machine. If any of these has a destination (architectural) register, then the mapped physical register is freed. These registers can be freed because the instructions that depend on them are also removed from the machine.

Although precise exceptions provide a simple programming model, a significant hardware cost may be incurred, and this cost may impact the cycle time of the processor. To simplify the design, the exception model can be made less precise, as was done in Digital's Alpha AXP architecture [Sites 1993], but in so doing, the process of recovering from an exception is made more complicated. For dynamically-scheduled processors, a relaxed exception model can contribute to shorter lifetimes of architectural-to-physical mappings thereby allowing physical registers to be reused more frequently and hence allowing an increased number of in-flight instructions. In the following discussion, memory operations are also assumed to be imprecise, and thus, the imprecise model used in this dissertation is even more imprecise than the model defined in Digital's Alpha AXP Architecture.

To facilitate the discussion, consider the scenario in which an architectural register R_a is named by an instruction I_1 as its destination register, and R_a has been mapped to a physical register R_p. With imprecise exceptions, the physical register R_p can be freed when:

1. Instruction I_1 has been *completed*.

[1]This statement is actually only partially true since the conditions given do not address changes in state caused by the execution of stores. To allow the recovery of the machine state, a non-merging buffer is required to hold the write data until the store instruction is committed. Only at this point can the data be written into the cache or the write buffer.

2. The instructions that use register R_p have been *completed*.

3. The architectural-to-physical mapping is killed by the *completion* of *any* instruction I_x that follows instruction I_1 in the program order if instruction I_x has register R_a as its destination *but only when all the branches preceding instruction I_x have completed*.

These conditions differ from those for precise exceptions in several important areas, and these are indicated by the emphasized typeface in the above list. First, conditions 1 and 2 are not subsumed by the architectural-to-physical mapping condition (condition 3), a result of it only being necessary for instructions to have been "completed" rather than have been "committed" (condition 2). Second, it is important for all preceding branch instructions to have been completed rather than for all preceding instructions to have been committed. And third, the writer of a physical register can cause the killing of *any* mappings created by preceding instructions, rather than only *the* preceding mapping. Taken together, these differences allow registers to be freed earlier, and allow the exact state of the machine to be recovered without assistance from software when a mispredicted branch does occur.

The performance and the register-file design impact of these two exception models are examined in Chapter 4.

Note that, unlike under precise exceptions, under imprecise exceptions, special consideration is required for asynchronous exceptions, such as those initiated by input/output devices. While asynchronous exceptions are not considered in this dissertation, one method for incorporating them into the imprecise exception model is as follows. When an asynchronous exception is detected, the processor first examines all completed instructions and selects the one, say instruction I_l, that is latest in program order and for which all preceding branches have completed. The processor then continues executing instructions until it commits instruction I_l, while, at the same time, neither issuing nor completing any instructions that follow I_l in program order. Once the processor commits instruction I_l, it calls the exception handler to process the exception, saving the address of the instruction following instruction I_l so that the application can be restarted once the exception has been processed.

2.3 Hardware-based Prefetching Using Stream Buffers

In this section, an enhanced stream-buffer model is proposed and the novel *incremental-prefetching* and *per-load stride prediction* techniques are introduced. Both the enhanced model and the novel techniques were developed as part of this dissertation. This section also describes how the enhanced model was integrated into the system model, and thus, how it could be integrated into a dynamically-scheduled processor.

The proposed stream-buffer model addresses a number of deficiencies in the original stream-buffer model proposed by Jouppi [1990]. These deficiencies have negative performance implications for dynamically-scheduled processors and potentially for statically-scheduled processors as well. This section also discusses techniques proposed by other researchers that can be used to address some of these deficiencies.

The examination of the proposed stream-buffer model begins in the next section with a discussion of prefetching and a description of the original stream-buffer model. Then, Section 2.3.2 discusses the deficiencies of the original model and the techniques that can be used to address them. Finally, Section 2.3.3 describes the implementation details of the proposed model.

2.3.1 Prefetching Fundamentals

The goal of prefetching is to bring data closer to the processor before the processor requires it, rather than waiting for a cache miss to occur before initiating a fetch for missing data, as is the case with non-blocking loads. The prefetch is initiated by some triggering event. With software prefetching, the trigger is prefetch instructions that are inserted into the code by a sophisticated compiler or user [Callahan 1991, Mowry 1992, Luk 1996]. With hardware prefetching, hardware is used to determine when a prefetch might be useful. Software prefetching has been most successful on codes with predictable access patterns, while hardware prefetching can be used with all types of applications (including the operating system). The reader is referred to [Mowry 1994] for a discussion on software prefetching techniques.

Examples of hardware prefetching techniques include Chen and Baer's lookahead PC reference prediction method [1992] and stream buffers [Jouppi 1990, Palacharla 1994]. The lookahead scheme uses a reference-prediction table, which is organized as a fully-associative buffer, to keep track of the data access patterns of load and store instructions. A prefetch is initiated for a memory instruction when the hardware predicts that the instruction will soon be executed. This prediction is done using branch prediction and a lookahead program counter. In each cycle, the hardware increments this counter by one, and then searches the branch-prediction and reference-prediction tables to determine if either table contains information about an instruction with the predicted address. If a match is found in the branch-prediction table, the lookahead counter is assumed to point to a branch instruction, and the counter is modified to take into account the predicted target address of the branch. However, if a match is found in the reference-prediction table, the address is assumed to point to a memory instruction. If the table contains sufficient information for this instruction, a memory request is issued using the stored information to determine the fetch address. When the prefetched data is returned, it is written into the data cache.

Stream buffers are less invasive than this lookahead scheme. The lookahead scheme is complicated by the need for additional ports into the data-cache tags when used in a superscalar processor. In such a processor, the cache-tag ports constitute one of the more critical resources, and increasing their number may result in a physically larger cache and/or a slower cache, or may be infeasible. Stream buffers, on the other hand, sit on the memory side of the data cache and while they must be probed on every data reference, they do not need access to the cache tags. Furthermore, stream buffers do not pollute the cache because the prefetched data is placed in special buffers.

In the original stream-buffer model proposed by Jouppi, stream buffers consist of a number of entries that are managed as a first-in-first-out queue. Each entry in the queue can store a block of data and the corresponding block address (i.e., tag). All entries of the stream-buffer queues are probed at the same time as the data cache probe is done. If the data cache probe results in a hit, the stream buffers are not touched. However, if a data cache miss occurs, and the desired block is in a stream-buffer entry, the cache block is read out and it is written into the cache. The stream buffer then deletes the matching block and those that precede it in the queue, and issues a prefetch request to the next lower level in the memory hierarchy to fill the empty entry(ies) with subsequent cache block(s) to the one that matched. When a miss occurs in both the cache and the stream buffers, a request for the block containing the miss address is issued to the next level in the memory system. Then, a stream buffer is allocated and told to begin fetching blocks subsequent to the missing block. Because each block that is fetched has a block-address one greater than the last, the stream buffers are said to use a *unit stride*. Once the request for the first block to be prefetched is launched into the memory subsystem, the stream buffer can issue another prefetch request if there remain empty entries in the queue.

To simplify the discussion of the proposed stream-buffer model and how it addresses the deficiencies of the original model, it is assumed that stores are implemented using write-around [Jouppi 1993] (i.e., no-write-allocate) and write-through policies with a write buffer situated between the data cache and lower levels in the memory hierarchy. However, the model can also be used with other store implementations.

2.3.2 The Proposed Stream-buffer Model

The proposed stream-buffer model addresses the following four deficiencies in the original stream-buffer model:

1. Overlapping prefetch paths,
2. Non-direct access to all of the stream buffer entries,
3. No memory-traffic filters, and
4. Unit strides.

The enhancements required to address these deficiencies are described below.

Overlapping prefetch paths: in the original model, when there are multiple stream buffers, a given block of data could reside in more than one stream buffer. Because this duplication reduces the effectiveness of the stream buffers, the original model was modified to ensure non-overlapping prefetch paths. To achieve non-overlapping paths, a comparator must be associated with each stream buffer entry. Although the comparators increase the design complexity, in practical systems they need to be included for enforcement of multiprocessor cache consistency anyway. Figure 2.6 shows an implementation of the proposed stream-buffer model in which there are four stream buffers with three entries each. Thus, for this implementation, 3 comparators are required per stream buffer, as illustrated in the figure.

Non-direct access to entries: in the original model, there is an access-time penalty to access data not in the entry at the head of the queue because it is first necessary to shift out the blocks closer to the head. The proposed stream-buffer model includes direct access to non-head entries in the queues to avoid this penalty and thus reduce the time required to read data stored in any entry.

No memory-traffic filters: in the original model, a stream buffer is allocated whenever a data reference misses in the cache and in the stream buffers. This allocation policy can result in prefetching down a subsequently unused stream should the data reference be isolated, thereby generating excess traffic to the memory system. One approach to addressing this deficiency is the technique proposed by Palacharla and Kessler [1994] called an *allocation filter*. This filter prevents a stream buffer from being allocated until the second miss to a stream is detected. A separate filter is required for each stream buffer; this filter is provided by the block labeled "memory traffic filters" in Figure 2.6. Section 2.3.3.2 describes how an allocation filter operates and how it could be implemented.

Another approach for addressing the problem of excess memory traffic is to limit the number of cache blocks fetched after a stream buffer has been allocated until the stream is found to be useful. The proposed

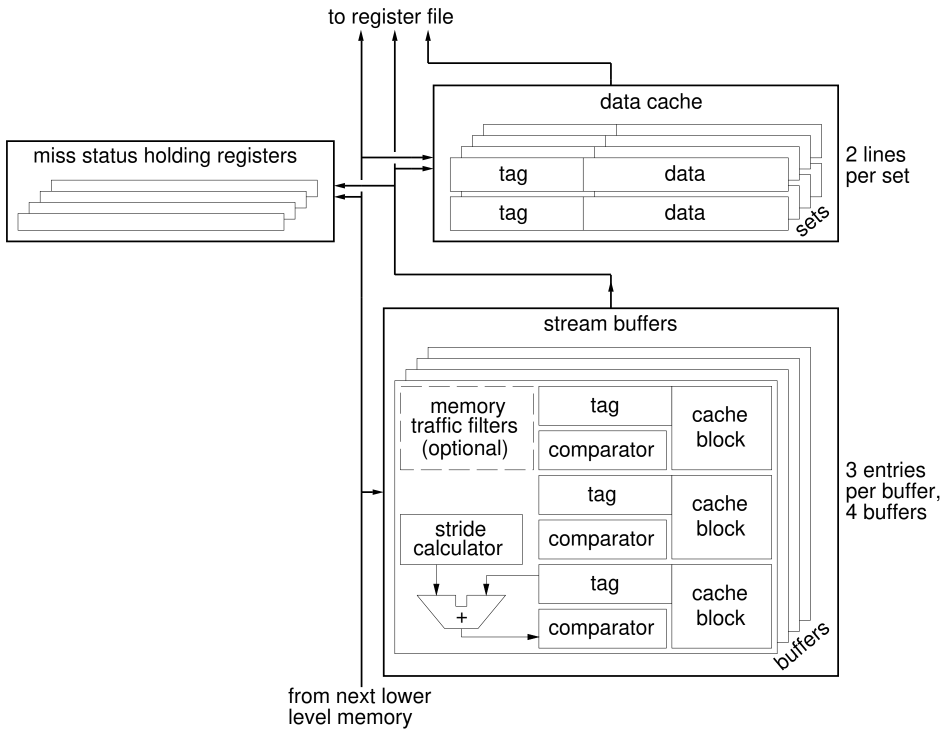


Figure 2.6: The proposed stream-buffer model integrated with the data cache and MSHR structure. A subset of the data paths are shown. In this example, there are four stream buffers with three entries each, and any one of the 12 entries can be used to supply the missing data for a cache miss.

incremental prefetching technique provides this functionality. To understand this technique, suppose that a stream buffer issues a prefetch request for a cache block B, and that this block is subsequently used to resolve a cache miss M. With incremental prefetching, if M is the n^{th} miss resolved by the stream buffer since it was allocated, the buffer is allowed to fetch at most the next 2^n blocks following B in the stream. This restriction on the number of blocks that can be prefetched remains in effect until 2^n is greater than N, where N is the number of entries in the stream buffer. At this point, the prefetching is controlled only by the availability of free entries, as in the case of a stream buffer not having the incremental prefetching capability. Incremental prefetching is described further in Section 2.3.3.1.

Unit strides: in the original model, prefetch strides were limited to the unit stride, thus potentially reducing the effectiveness of the stream buffers and generating excess memory traffic. Instead, the stride can be determined dynamically using previous miss addresses. These addresses are the addresses of data items that were not found in the data cache or in the stream buffers. The miss addresses are typically maintained in a table. A stride prediction is made by selecting one of the addresses stored in this table, and subtracting the address of the current missing data; the difference is then used as a stride. The proposed *per-load stride predictor* records the last miss address for N static loads. In other words, each miss address is associated with a program-counter address. Thus, a stride prediction is based only on the past memory behavior of the static load for which the prediction is being made. The *per-load stride predictor* is based on the stride-directed prefetch technique proposed by Fu et al. [1992] for preloading the data cache. Section 2.3.3.3 describes the per-load stride predictor and an enhancement to implement an allocation filter.

Palacharla and Kessler [1994] have proposed a second type of dynamic-stride predictor. With this type of predictor, the stride prediction for a load instruction is based on the last N miss addresses, rather than, as in the case of the per-load stride predictor, only the last miss address for the load in question. Thus, their stride predictor captures cache-miss patterns that originate from the access patterns of more than one static load. Section 2.3.3.2 describes the implementation of such a stride-predictor, which is based on their *minimum delta* scheme.

2.3.3 Implementation of Model for a Dynamically-Scheduled Processor

This section describes an implementation of the proposed stream-buffer model for the system model and for a dynamically-scheduled processor. This implementation was developed in the course of the research described in this dissertation. To simplify the discussion, the implementation details of allocation filters and dynamic-stride predictors are described in separate subsections.

Stream buffers reside on the memory side of the data cache, as illustrated in Figure 2.6. On each data cache reference, the stream buffers are probed along with the data cache, and if the data-cache probe results in a miss, an MSHR is allocated. The MSHR is required in most implementations for ensuring that secondary misses are detected while the cache block is being forwarded to the data cache from either the stream buffer or the next lower level in the memory hierarchy. An address-stack entry is also required to ensure that the named register is loaded when the block is returned.

If a cache miss should occur, one of four sequences of events can occur depending on the result of the stream buffer probe. These sequences are described below.

Data is Present

One possible outcome of the stream buffer probe is that the tag matched and the data is present. In this case, in the next cycle, both the cache line to which the miss address maps and the target register of the load are written. If this line is required by another load access, the data can be bypassed to the load at the same time the cache line is written. When a hit occurs, the matching entry is shifted out of the corresponding stream buffer. Because the stream buffer is a FIFO queue, this shifting results in the invalidation of the matching entry and all entries from the entry at the head of the queue to the matching entry. Invalidating an entry entails clearing the valid bit associated with the tag. This invalidation occurs *whenever* an entry's tag matches and the valid bit is set.

Because the stream buffer now has at least one invalid entry, it can begin prefetching again along the same path. The first step in initiating a prefetch is to determine the address of the next block to be fetched into the buffer. If the buffer contains no valid entries, the block address of the next block to be fetched will be one more than the tag of the matching entry. If, on the other hand, the buffer contains some valid entries, then the block address of the next block is one more than the tag contained in the bottom-most entry in the buffer.

Once a block address is determined, the next step is to check to see if the block to be prefetched is present in another stream buffer or is being prefetched. This step is necessary to ensure that the stream buffers are prefetching down non-overlapping paths. This check is performed by doing an associative search of all valid stream buffer entries. If a match is found, the stream buffer is prevented from fetching further down the present path, and since it is partially full, the stream buffer is flagged to be used next when a stream buffer is needed to follow a new path. In this way, inactive stream buffers are reallocated before active ones. As a result of ensuring non-overlapping paths, it is possible for a stream buffer to remain empty until it is reallocated on a stream buffer miss.

If the tag of the to-be-prefetched block is not already in another stream buffer, the tag is written into the bottom-most entry in the buffer, the valid bit for the tag is set, and a fetch request is issued to the memory. When the memory launches a fetch request from a stream buffer, the launched bit for the entry is set. After launching a fetch request, if the stream buffer contains some invalid entries, the just described steps for initiating another prefetch are applied.

Fetch Request for Data Issued

The second outcome of the stream buffer probe is that a stream buffer has issued a fetch request for the desired block, the memory system has started processing the request, but the data has not yet been returned. As described above, on a hit, the matching entry is shifted out of the stream buffer. When a block is returned from the memory, its return may be due to either a cache-miss initiated fetch or a stream buffer initiated prefetch. To determine which component is awaiting its return, an associative search is done of all the valid MSHRs and all the valid stream buffer entries. If this search reveals that there is at least one matching MSHR, the block is written into the cache and all waiting registers are filled. If this search also reveals that there is a matching stream-buffer entry, the data is written into the entry, and the present bit is set. Note that it is possible for both searches to result in no matches because: (1) a mispredicted branch may have been detected between the time that a cache miss occurred and the request block returns from the memory system and this misprediction caused the initiating load instruction to be flushed from the system, or (2) the stream buffer that initiated the cache-block fetch may have been reallocated before the fetch completed.

Matched to an Unlaunched Entry

The third outcome of the stream buffer probe is that there is an entry with a matching tag but the memory request for this block has yet to be launched. As in the above two cases, the matching entry is shifted out of the stream buffer. However, unlike the previous case, a fetch request is issued to the memory. Because the data being fetched is required to resolve a data-cache miss, this fetch request is serviced before any requests that are associated with stream-buffer prefetches.

Stream buffer miss

The final outcome of the probe is that there was no valid match. In this case, the address of the subsequent block is calculated (as in the first outcome when there were no remaining valid entries) and a probe is done of all valid stream buffer entries. If a match is not found, a stream buffer is chosen to begin prefetching down this new path. A stream buffer is chosen from among those which were prevented from following a path (because the same path was being followed by another buffer), or if there are no such stream buffers, then the stream buffer with the oldest access-time stamp is selected. The access-time stamp is set whenever a stream buffer probe results in a hit and when the stream buffer is allocated.

2.3.3.1 Incremental Prefetching

This section describes the proposed novel prefetching technique called incremental prefetching. Incremental prefetching reduces the memory bandwidth consumed by stream buffers through limiting the number of blocks fetched for a stream to 2^{m-1} where m is the number of cache misses resolved by the stream buffer including the miss that led to the allocation of the stream buffer. Thus, on allocation ($m = 1$), a stream buffer will fetch one block, and if this block is used to satisfy a subsequent cache miss ($m = 2$), it is allowed to fetch 2 blocks, and so on. A block can satisfy a cache miss if it is present or if an entry has been assigned to it, irrespective of whether the fetch request has been launched.

To implement incremental prefetching, a *fill-me* bit is associated with each entry in the stream buffer. The fill-me bit indicates to the prefetching hardware whether it should launch a fetch into the memory system for the stream buffer entry. The fill-me bits of all entries are cleared when a stream buffer is allocated. The bit of the head entry is then set while the address of the first block to be fetched is written into the entry. As the fill-me bit is set, the prefetch controller will launch a fetch request for the block of data. If this block is used to satisfy a subsequent cache miss ($m = 2$), the fill-me bits of the next two entries are set. The prefetch hardware will then launch fetch requests for these two entries, using the protocol described in the preceding section. Should either of these two entries be used, then the next four fill-me bits are set. This process continues until all the fill-me bits have been set, at which point prefetching is controlled only by the availability of free entries.

2.3.3.2 Allocation Filters and the Minimum-delta Stride Predictor

The allocation filter and minimum-delta stride predictor proposed by Palacharla and Kessler [1994] can also be integrated in to the proposed stream-buffer model. They are built around a *miss-address table* which is implemented as a ring buffer with N entries. This table is used to record the address of the data required by the last N loads that missed in both the cache and the stream buffers.

Prefetch filters prevent a stream buffer being allocated until a second or third miss occurs along the same prefetch stream. If a reference to an address A, with a corresponding block address A_b, misses in the cache and in the stream buffers, a check is done to see if a block with block address $A_b - 1$ is in the miss-address table. If $A_b - 1$ is present, then the reference to block A_b represents a possible second miss along the same path, and a stream buffer is allocated with a unit stride if this path does not overlap an existing prefetch path. The stream buffer will then begin to prefetch blocks with block addresses $A_b + 1$, $A_b + 2$, etc. Regardless of whether the block $A_b - 1$ is present in the miss-address table, the address A is written into the table after the table is probed. To write an entry into the table, first the write-head pointer is shifted to the next entry in the buffer, and then the corresponding entry is written. At the same time as the address A is written into the table, a fetch request is sent to the memory for the missing cache block at block address A_b.

Non unit-stride prefetching is more complicated as it requires first determining a stride and then determining whether to allocate a stream buffer. On a stream buffer miss, the allocation filter is applied to determine whether a unit-stride should be used. If there is a filter miss (i.e., the address A is not part of a unit-stride stream), a calculation is done to determine the minimum unsigned difference M_u between the address A and the N addresses $A_1, \ldots, A_N$ stored in the miss-address table, that is, $M_u = min(\|A - A_1\|, \ldots, \|A - A_N\|)$. If this minimum is zero, a cache-line-fill request is issued for the block containing A, no stream buffer is allocated, and A is not written into the table.

On the other hand if the minimum is not zero, then the first address A_f of the prefetch stream is determined, and this value is used to determine if a stream buffer should be allocated. For notational convenience, assume that the value of M_u corresponds to the difference between A and the stored address A_i ($i \leq N$), that is, $M_u = \|A - A_i\|$, and let $M_s = A - A_i$. With this notation in mind, $A_f = A - 2 \times M_s$, because, if A is the third miss in the stream, $A - M_s$ and $A - 2 \times M_s$ would have been respectively the second and the first misses in the stream. If this miss address A_f is in the same cache block as A, then a stream buffer is not allocated because the stream has yet to cross a cache-block boundary. The address A is, however, written into the miss-address table to permit the detection of the stream should it cross a cache-block boundary. If, on the other hand, A_f is not in the same block as A, the table is first searched for a cache block containing the address A_f, and then A is written into the miss-address table. If such a cache block is found, the address A is assumed to be the third miss to the stream, and a stream buffer is allocated. This stream buffer will use a stride S equal to M_s if M_u is larger than the size of a cache block, and, otherwise, a unit stride with the same sign as M_s. The stream buffer then fetches addresses $A + S$, $A + 2 \times S$, etc. If S were permitted to be smaller than the size of a cache block, at some point the just-fetched address and the next-to-be-fetched address would correspond to the same cache block, and the stream buffer would be halted because its own prefetch path overlaps itself.

2.3.3.3 Per-load Stride Prediction

The per-load stride predictor can also be integrated into the proposed stream-buffer model. This predictor is based on the stride-directed prefetching scheme proposed by Fu et al. [1992]. It consists of a fully-associative buffer with N entries. This buffer is referred to as the *missed-load buffer*. Each entry is used to store the program-counter (PC) address, the previous miss address A_s of a static load instruction, and optionally, the previously-calculated stride S_s.

Assume for the moment that filtering of stream-buffer allocations is not supported, and thus, the field for S_s is not required. Now, suppose a load instruction L is reading data from an address A, and this data is not in the data cache or the stream buffers. In this case, a memory request is issued for the cache block containing address A, and a probe is done of the missed-load buffer using the program-counter (PC) address of L. If a matching entry is not found, one is allocated, and the PC address[2] and the miss address A are written into it. Because there is no matching entry, a unit stride is assumed, that is the block stride equals 1, and the address stride S equals the size of a cache line. If, on the other hand,

[2]The PC address is assumed to be recorded in the active-list entry belonging to each load instruction as it is inserted into the dispatch buffer (see Section 1.1.1).

a matching entry is found, the stride is calculated as the difference between the miss address and the previous miss address A_s, that is, $S = A - A_s$. After calculation of the stride, the address A is written into the appropriate entry. A check is then done to see if the address $A + S$ overlaps the prefetch paths of any of the stream buffers. If it does not, a stream buffer is allocated with stride S, and it begins prefetching at address $A + S$.

To support the filtering of stream-buffer allocations, the third field is required for storing the previously-calculated stride S_s for a load. If the probe of the missed-load buffer does not locate an entry for the load, the same steps as outlined above are performed, with the addition that when an entry is allocated, the field for the previously-calculated stride is set to zero. The next time the same load misses in both the stream buffers and the data cache, after calculating the new stride S using the saved miss address A_s, this stride and the new miss address A are written into the appropriate entry. No stream buffer is allocated, however. The third and subsequent times the same load misses in both the stream buffers and the data cache, again the new stride S is calculated, but this time an attempt will be made to allocate a stream buffer if the saved stride S_s equals the new stride S and neither equals 0. Irrespective of whether a stream buffer is allocated, the stride S and miss address A are written into the appropriate entry.

Thus, the first miss (with address A_1) results in a unit-stride allocation, the second miss (with address A_2) results in no stream buffer allocation, the third miss (with address A_3) results in a stream buffer being allocated if $S = A_2 - A_1 = A_3 - A_2, S \neq 0$, the fourth miss (with address A_4) results in a stream buffer being allocated if $S = A_3 - A_2 = A_4 - A_3, S \neq 0$, and so on. Note that, as with the minimum-delta stride predictor, if the stride S is smaller than the size of a cache block, then a unit stride is used with the same sign as S. As before, this strategy guards against the prefetch stream of a stream buffer overlapping with itself.

2.4 Chapter Summary

In this chapter, the implementation frameworks for dynamically-scheduled processors have been described with a focus on the techniques that are used to provide the functionality described in Section 1.1.1. In addition, the results of an analysis of several important aspects of these frameworks have been presented along with the enhancements to one of these frameworks that followed from the analysis; the analysis was performed as part of this dissertation.

An examination has been presented of existing processors and the functional similarities between the approaches used by these processors to implement dynamic scheduling. This examination emphasized two unique properties of the approach used in the MIPS R10000: (1) a clear distinction between the hardware registers and the other system components, and (2) an absence of any distinction between the registers which provide the functionality of physical registers and those that provide the functionality of rename registers. Owing to these properties of the R10000 approach, a processor model similar to the R10000 was adopted for the research described in this dissertation. The exception models implemented by this processor model were then more formally discussed in the second part of the chapter. This discussion focused on the conditions under which the physical registers could be freed for reuse. Finally, in the third part of the chapter, a stream-buffer model for implementing hardware-based prefetching was presented. The examination of this model emphasized the improvements this model makes to the original model. In addition, it emphasized how this model could be incorporated into a dynamically-scheduled processor and how it was incorporated into the system model used in this dissertation. The system model is described in the next chapter.

Chapter 3

System Model and Modeling Methodology

This chapter describes the system model developed for the investigation into the memory-system design requirements of dynamically-scheduled processors that is presented in this dissertation. The chapter also describes the methodology used to identify and select the details that were included in the model.

Two important aspects to the modeling of any computer system are: (1) including the appropriate details so that the simulation results are realistic, and (2) excluding details so that generalizations can be drawn from the results. The goal of this chapter is to provide insights into the design of a system model, and the tradeoffs that must be made between accuracy and generality.

The chapter beings with a discussion of the relationship between accuracy and generality. Section 3.2 follows with an explanation of why the register file is the focus of the research presented in this dissertation. Section 3.3 then discusses the factors that affect the design of the register file and the demands placed on it by the lockup-free cache, the stream buffers, and the memory interface. Section 3.3 also discusses the factors that are excluded from the system model to achieve a balance between accuracy and generality. The system model is then described in Section 3.4, with the technique used for simulating the execution of an application on such a system described in Section 3.5.

3.1 Trading Accuracy for Generality

Accuracy and generality are competing objectives. Accuracy describes how closely the model emulates the behavior of a real (i.e., actual) system with the same functionality as is embodied in the model. A very accurate model is useful for estimating the performance of a very specific point in the vast design space of systems. Generality, on the other hand, is a measure of how well the results obtained from the model represent real processors with different implementation characteristics. A less accurate and more general model is useful for discovering performance trends and design relationships at various points in the design space. These trends and relationships may then be used to select a more limited number of design points at which further investigation is warranted.

Competition between accuracy and generality is due to the central role time plays in determining the behavior of dynamically-scheduled processors, a role which is a by-product of executing instructions in parallel. While instructions that are executed in parallel are dependent on each other by virtue of having to share hardware resources, all instructions are dependent on each other by virtue of the order in which instructions must be fetched and committed. Thus, the execution of any instruction can affect the temporal relationship between all future instructions. For example, a cache miss may delay the issuing of instructions that are dependent on the value being loaded, thereby delaying the completion of these instructions, and thus temporarily reducing the rate at which unfetched instructions can be fetched and inserted into the processor. As a result, the pool of instructions that are in some stage of execution may be affected, perhaps leading to perturbation of the instruction-issue order, the branch prediction accuracy, and the cache hit rate.

Because time cannot be ignored, it is not possible to accurately investigate a processor-based system by accurately modeling and investigating, in isolation, the components of the system. The model used for an investigation must balance the desire to accurately model certain features of a specific system and the desire to be representative of systems in general. The model used in this dissertation was designed to allow the identification of performance trends and design relationships, and as such, does not correspond to a specific system.

3.2 Importance of the Register file

This dissertation describes an investigation of the complexity-performance relationship in dynamically-scheduled processors and the impact complexity has on algorithms for (static) instruction scheduling. The register file was chosen as the focus of this investigation for two reasons.

First, a wide-issue dynamically-scheduled processor is likely to have a number of components that are instrumental to its operation and significantly impact performance. These components include the register file (either implicitly or explicitly implemented), the dispatch buffer (i.e., a reorder buffer or a dispatch queue), and the register renaming unit. The implementation size and complexity of these structures tend to scale together since it is desirable that no one structure offers a disproportionate amount of functionality. For example, it would be inappropriate to employ an 80 entry reservation station in the Pentium Pro because, with a 40-entry reorder buffer, at most 40 of the reservation-station entries could ever be concurrently used. The register renaming unit and the dispatch queue are subject to wide variations in implementation architecture and circuitry as suggested by the discussion in Section 2.1, while the register file design space is more limited. The functional design of a register file can be described by the clock cycle count required to read or write a register, by the number of registers, and by the number of read and write ports.

Independent of how the dispatch buffer and renaming unit are implemented, they will have structures similar to those found in the register file (such as numbers of ports or numbers of entries). Since an objective of the work described in this dissertation was to balance the competing objectives of accuracy and generality, the register file was selected as the focus. For this same reason, a processor model was chosen that implements dynamic scheduling using the less implementation specific framework of a single dispatch queue, a register file which holds both speculative and non-speculative results, and a mechanism for differentiating between the two. This framework offers a clear separation between the register file and other components while maintaining the impact each has on the design of the others. The issue of balance of accuracy and generality is discussed further in Section 3.3.1.

A second reason for focusing on the register file is the following. Although many other researchers have investigated dynamically-scheduled processors that use register renaming, there is no known research that has focused specifically on issues affecting the register file. And, for the most part, in the literature describing these investigations, many authors have neglected to state how many physical registers were available for the renaming of architectural registers. An exception is an investigation carried out by Wall on the limits of instruction-level parallelism. This investigation included an examination of the impact of varying the number of registers for a 64-issue, 2048-instruction window machine with unit operation latencies [Wall 1993]. Bradlee, Eggers, and Henry investigated the performance tradeoffs of the number of registers for a RISC instruction set architecture with various kinds of compiler support, but this study was for a statically-scheduled, single-issue processor [Bradlee 1991]. Franklin and Sohi also considered a statically-scheduled, single-issue processor in their study of register life times and the replacement of the register file with a distributed mechanism [Franklin 1992].

3.3 Factors Affecting the Design of the Register File

This section describes the factors that have a direct impact on the design of the register file. In the process, this section also examines the relationships between the register file, and the other components

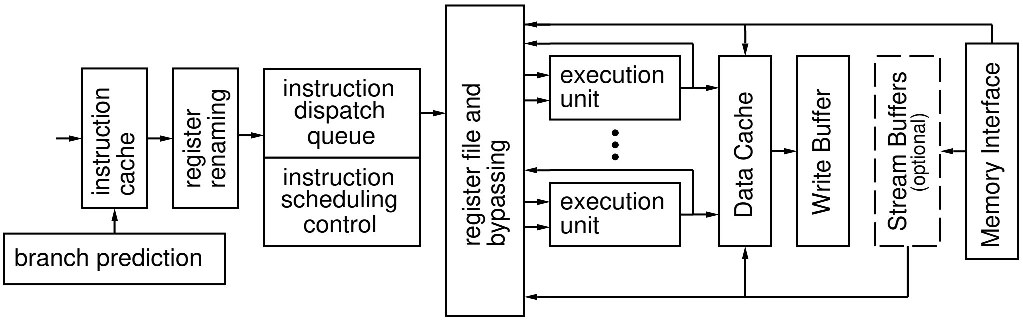


Figure 3.1: Overview of the dynamic scheduling implementation showing only the data path.

in the memory system used to service cache misses.

Stated simply, the performance of an application is dictated by the amount of time it takes for it to be processed. On a superscalar processor, the time required to process the instruction stream of an application is determined by (1) the time required to execute each instruction, that is the instruction latency, and (2) the number of instructions that can be executed in parallel. One factor that influences instruction latency is the cycle time of the register file since most instructions will both read operands from the register file and write results to it. It is therefore perhaps not surprising that many (most) processors can read and write a register in a single cycle. Single-cycle accesses are also preferred because larger access times require deeper pipelines, and deeper pipelines may induce more hazards, larger branch penalties, and more complex hardware for hazard detection and data forwarding. Given the popularity of single-cycle accesses, the processor model described here assumes a single-cycle access; the performance impact of the period of a cycle is discussed in Section 4.4.

The instruction-level parallelism obtained by an application is affected by the availability of physical registers for renaming since most instructions require a destination register, and an instruction cannot be considered for execution until one can be assigned to it. When there is no free physical register for an instruction, the process of inserting instructions into the dispatch queue must be stalled. The dispatch queue and register renaming hardware are shown in Figure 3.1 along with the other components of the processor model.

Registers can be freed only when doing so will not prevent the processor from recovering and resuming execution after an exception occurs. With *precise exceptions*, it is required that the instructions preceding the faulting instruction in the fetch order be allowed to change the state of the system while those following the instruction will not. In this exception model, a register cannot be freed until all the instructions preceding the instruction writing this register are guaranteed to have been completed. Under *imprecise exceptions*, the state of the system is not maintained so exactly, thereby allowing registers to be freed earlier, and hence allowing for their more frequent reuse. Although machines with truly imprecise exceptions are now rare in general purpose systems (because it prohibits multiprogramming and modern OS systems), a true imprecise exception model was examined as a best case limit for other hybrid exception approaches [Patterson 1995]. Exception models are discussed in more detail in Section 2.2.

The freeing of registers is also affected by branch prediction. Branch prediction is typically used in dynamically scheduled processors to allow the processor to move instructions across branches and thereby increase the pool of those instructions available for issue. Branch prediction, however, can negatively affect performance in two ways. First, mispredicted branches result in the execution of unnecessary instructions, thus reducing the average useful instruction-level parallelism. And, second, as discussed above but in regard to exceptions, because the direction a branch takes is not definitively known until it is executed, the physical registers that are written by instructions following the branch in program order cannot be freed until the branch is executed. Hence, the time that a register is live (i.e., in use) depends on the accuracy of the branch prediction hardware.

Another factor that directly affects the time that a register is live is the miss rate of the primary data cache and the time required to service cache misses. When an instruction does not find the required data in the cache, the instruction's completion is delayed until the data can be fetched. If the instruction is a load instruction, then the register that is the target for the load will need to remain live until the fetch can be completed. In addition, the miss will delay the issuing of any instructions in the dispatch queue that *require the result of the load*, hence keeping the registers assigned to them live for longer. Finally, because the servicing of a data cache miss requires hardware to track the outstanding primary and secondary misses, if there are insufficient hardware resources available, memory instructions cannot be issued. Thus, memory instructions that are not data dependent on other, already issued, memory instructions may be prevented from issuing, and thus, may increase the lifetime of the physical registers. This increase may be offset with the use of a hardware-based prefetcher. The goal of prefetching is to reduce the average time required to service a cache miss.

The number of physical registers and the frequency of their reuse have a significant impact on system performance since most instructions require a destination register and instructions cannot be inserted into the dispatch queue when there are no free registers. Instruction-stream stalls may result in the hardware not being able to keep the dispatch queue full, thereby reducing the number of instructions available for selection by the scheduler. In turn the result may be a limit on the ability of the scheduler to schedule the maximum number of instructions per cycle.

As mentioned above, the register reuse frequency is a function of the exception model, the branch prediction accuracy, and cache misses. It is also a function of the issue width, and the number and type of functional units, for these factors affect the length of time between the insertion of an instruction into the dispatch queue and its completion. Chapters 4 and 5 consider the demands that these factors place on the number of registers and the number of register file ports, and how these factors affect the cycle time of the register file. Chapter 5 also considers the relationships between the memory-system components that are used to service data-cache misses.

3.3.1 Trading Accuracy for Generality in the System Model

The factors described in the previous section were chosen as they have a direct impact on the design of the register file, and, as such, they must be included for accuracy. Other factors have a less direct impact, leading to the hypothesis that they can be excluded to increase the generality of the system model. This section discusses the factors that were excluded and how their exclusion reduces the accuracy of the model.

One source of performance degradation of a dynamically-scheduled processor is stalls of the instruction-fetch stream, which can be caused by instruction-cache misses and by the inability to resolve the address of a predicted-taken branch or jump instruction. These stalls reduce the supply of new instructions to the processor, and thereby reduce the number of instructions that might be in the system at one time. Thus, while a stall is in progress, fewer physical registers are required for remapping destination registers. At the same time, however, more physical registers may be required since servicing of instruction-cache misses may increase the effective latency of data-cache misses. Moreover, if stream buffers are used, the servicing of an instruction-cache miss may interfere with the prefetching of data, thereby reducing the effectiveness of stream buffers. To avoid these potential performance impacts of instruction-stream stalls, it is assumed that (1) instruction-cache misses do not interfere with the servicing of data fetches, and (2) all branch instructions aside from conditional branches are 100% predictable. In addition, owing to limitations of the simulation infrastructure, which is described in Section 3.5, neither arithmetic exceptions nor multiprogramming are modeled. Furthermore, virtual-to-physical address translation is not modeled.

Another source of delays that affect the fetching of data is the flushing of data from the write buffer to either the data cache or the next level in the memory hierarchy. A write buffer may also affect performance because a stall in the issuing of store instructions is required if there are no free entries in the buffer. Such a stall will delay the issuing of memory instructions later in fetch order that read or write the same location, and will delay the committing of instructions, thereby increasing the lifetime of the physical registers. To avoid these performance impacts, a write buffer is assumed to consume no memory bandwidth and to induce no memory-instruction stalls; the details of the assumed write-buffer implementation are given in Section 3.4.3.

The lifetime of physical registers is also affected by the order in which ready-to-be issued instructions are selected for issue. In systems using the precise-exception model, for example, it may be advantageous to issue the earliest instructions in fetch order first so as to decrease the time required to commit an instruction and thereby free up the allocated physical registers. However, this approach requires that the hardware keeps track of the fetch ordering of instructions in the dispatch queue, and this functionality increases the hardware complexity [Johnson 1991]. This additional complexity can be avoided by using random selection, but at the cost of a possible degradation in performance. On the other hand, if more hardware complexity can be tolerated, other approaches could be used that, for example, decrease the time required to perform the computation on the critical path. Since the performance-functionality tradeoff has been explored by Butler and Patt [1992] and since the hardware complexity of an earliest-first approach is not excessive, the processor model used in this work assumes this approach.

3.4 System Model Details

The previous section summarizes the factors that are accurately modeled and those that are not. In this section, further details are given of the system model.

The model implements a RISC superscalar processor with an instruction set based on the DEC Alpha instruction set [Sites 1992]. The processor supports non-blocking loads and non-blocking stores, and allows all instructions to be speculatively executed. The processor includes separate instruction and data caches. The instruction cache does not interfere with memory accesses for data and therefore the instruction cache has a fixed cache miss penalty. The data cache can be configured to be either lockup or lockup-free; the lockup-free cache can support a configurable number of outstanding cache misses. In addition, the system model can be configured to include one or more stream buffers.

The instruction scheduling logic includes a single dispatch queue for all functional units, and its size is configurable. The dispatch queue is similar to the fast dispatch stack of Dwyer and Torng [Dwyer 1992]. In a clock cycle, the number of instructions that can be inserted into the dispatch queue is equal to 1.5 times the maximum issue width of the processor; this value was chosen to reduce the possibility of instruction insertion being a significant bottleneck. Instructions are selected for issuing using a greedy algorithm that issues the earliest instructions in the fetch order first, a policy also used in the MIPS

R10000 as well as in some of the other processors described in Section 2.1.

The issue logic includes hardware to dynamically disambiguate memory addresses so as to allow memory instructions to be issued before those occurring earlier in the fetch order. As the focus of this work is on factors directly affecting the register file rather than tradeoffs in the design of memory disambiguation hardware, it is assumed that sufficient hardware exists to allow disambiguation to be done across all memory instructions present in the dispatch queue. For a discussion of the tradeoffs in the design of memory disambiguation hardware, the reader is referred to Gallagher [1995].

The register file includes a configurable and equal number of integer and floating-point registers. The register renaming scheme is modeled after the scheme used in the IBM ES/9000 [Liptay 1992]. The simulator implements both precise and imprecise exceptions, with the only source of exceptions being mispredicted branches; arithmetic exceptions are not modeled. Registers are freed after the conditions of the assumed exception model are met, of which one is the commitment of an instruction. In each clock cycle, up to n instructions can be committed, where n is equal to twice the issue width of the processor. Thus, under precise exceptions, the maximum number of registers that can be freed in each clock cycle is also n. On the other hand, under imprecise exceptions, the number could be higher than n because the commitment of one instruction can initiate the freeing of multiple registers. See Section 2.2 for more details on the freeing of registers under each exception model. As in the MIPS R10000, recovery from a mispredicted branch requires one cycle, and mispredicted branches are recognized as soon as the misprediction is detected. Branch prediction is done using a scheme proposed by McFarling [McFarling 1993] that includes two branch predictors and a mechanism to select between them. This scheme is used to predict the direction of conditional branches; all other control flow instructions are assumed to be 100% predictable.

Figure 3.2 presents an overview of the system model described to this point. Additional details concerning the processor, branch predictor, non-blocking memory instructions, and the memory interface are in the remainder of this section, while details of the stream buffer implementation are given in Section 2.3.

3.4.1 Processor

The instruction scheduling rules and the functional unit latencies were chosen to resemble those of a number of commercial processors including the PowerPC 604 [Song 1994], the DEC 21164 [Edmondson 1995], the MIPS R10000 [Gwennap 1994a] and the SUN UltraSPARC [Agrawal 1994]. The processor can issue 4 or 8 instructions per cycle, which are issue widths representative of the current state-of-the-art

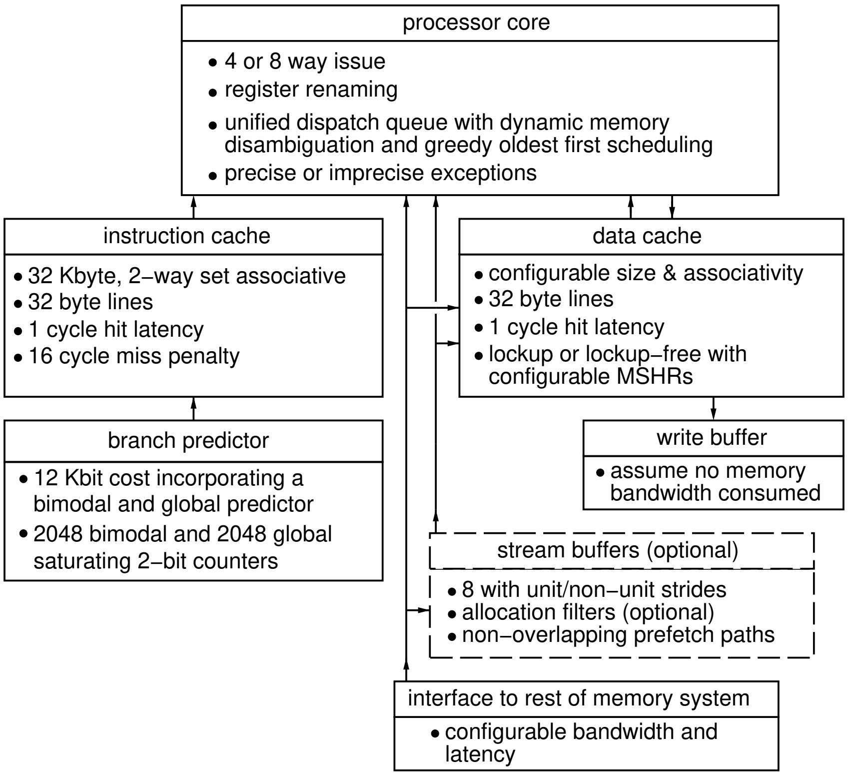


Figure 3.2: Overview of system model.

and future processors. For the four-way issue processor, each instruction word can contain at most four operations of which there can be at most: four integer operations, one floating-point division operation, two floating-point operations, two memory operations (i.e., two loads, two stores, or one of each), and one control flow operation (i.e., branch, subroutine call or return). The issue rules for the eight-way issue processor are the same but for each of the above instruction classes, twice the number can be issued in a cycle.

The processor's execution pipeline is assumed to have four stages: in the first cycle, an instruction is issued, in the second, it is provided with its source operands, in the third, it is executed, and in the fourth the result it generates is written to a physical register, or in the case of a store, to the write buffer. Forwarding is used to make results available to instructions later in fetch order at the end of the third stage. With the exception of the execution stage, all stages have a single-cycle latency. All integer functional units have single-cycle latencies except for the multiply unit, which is fully pipelined and has a six-cycle latency. All floating point units have three-cycle latencies and are also fully pipelined, with the exception of the floating-point divider. The floating-point divider is not pipelined and has an eight-cycle latency for 32-bit divides, and a 16-cycle latency for 64-bit divides. Finally, stores take one cycle to be resolved and there is a single load-delay slot. A scheme similar to that in the MIPS R10000 is used to permit an instruction that uses the result of a load to issue before it is known whether the data being loaded is in the cache. As explained in Section 2.1.4, an instruction I that uses the data being loaded by a load instruction will be aborted and later reissued if, in the cycle after issuing I, it is determined that the required data is not in the cache. By speculatively issuing I on the assumption that the load would hit in the cache, for cache hits a load instruction has an effective latency of two cycles.

3.4.2 Branch Predictor

The combined-predictor branch prediction scheme has a 12K bit cost and comprises a bimodal predictor and a global history predictor. McFarling found this choice of predictors to be most effective in his study [McFarling 1993]. The bimodal predictor employs the classical branch prediction idea of having a set of counters that indicate the direction taken by the branches that shared the counter the previous times they were executed; 2048 two-bit saturating counters are used. The global history predictor uses a shift register to generate a combined history of the direction of the last n branches. The contents of this register are exclusive ORed with the program counter word address to select one of another set of 2048 two-bit counters; these counters are used in the same way as the first set.

The selection mechanism, the *predictor predictor*, is essentially a bimodal predictor also containing 2048 two-bit saturating counters. Each counter reflects which of the two branch predictors has been correct most often for the branches that share the counter. A block diagram of the branch prediction scheme is shown in Figure 3.3.

The global-history shift register is updated after each conditional branch is inserted into the dispatch queue using the predicted direction. By updating the shift register during insertion, it is possible to take advantage of already identified branch patterns when determining the instruction to fetch next. A consequence of updating the register early, however, is that on a mispredicted branch the shift register must be loaded with the value it contained before the mispredicted branch was inserted into the dispatch queue. Thus, before each branch is inserted into the dispatch queue, a copy of the shift register is made.

The two-bit saturating counters are updated when a conditional branch is executed. To select the two-bit global-predictor counter that is updated, the saved value of the shift register is used rather than current value. The saved value is used to ensure that the two-bit counter used to make the prediction is the same one that is updated when the branch is executed. The bimodal counter and the counter for selecting the predictor are updated using the program-counter address of the branch instruction.

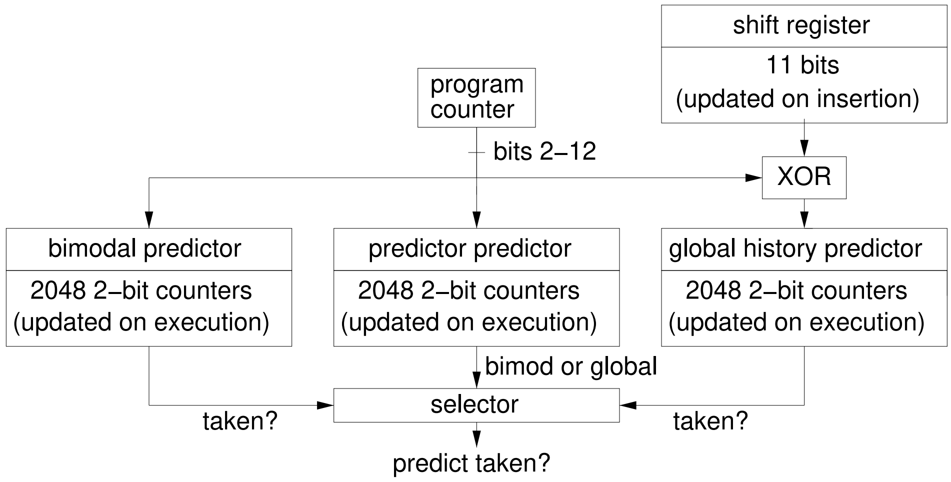


Figure 3.3: Branch prediction scheme.

3.4.3 Memory System for Data

As noted above, stores and loads are both non-blocking. Non-blocking stores are commonly implemented using one of two methods. The first method entails placing the data to be stored in a write buffer while the cache fetches the line into which the data is to be stored. The second method is to use write policies other than fetch-on-write, such as write-around, which neither fetch data on a write miss nor write the new data into the cache. Instead the data is written directly to the next lower level in the memory hierarchy [Jouppi 1993]. These methods do not require very complex hardware and are common in microprocessors. The system model assumes that stores are implemented using the latter method, namely, using write-around (i.e., no-write-allocate) and write-through policies with a write buffer situated between the data cache and lower levels in the memory hierarchy. To limit the impact of second order effects on the study of the factors that affect the design of the register file, no memory bandwidth is required to retire stores in the write buffer. This assumption prevents any stalls due to a full write buffer and prevents stores from delaying the servicing of cache fetches.

The data cache can be configured to be lockup or lockup-free. The lockup-free cache employs a MSHR (Miss Status Holding Register) organization that can support a configurable number of in-flight cache misses. For the four-way issue processor configuration, the register file has eight read ports and sufficient write ports to prevent any write-port conflicts arising when registers are filled on the resolution of a cache miss. For the eight-way issue processor configuration, the number of ports is doubled. (Section 4.4 discusses read and write ports further). The system model can also be configured with one or more stream buffers for use in prefetching data. The stream buffers may use either unit-strides or dynamically-calculated strides; stream buffers are described in Section 2.3.

Requests for blocks of data are sent via the memory interface to the next level in the memory hierarchy. The memory interface returns the requested block in a constant number of cycles, called the *fetch latency*. The bandwidth of the interface is constrained by controlling the number of cycles between the launching of fetch requests. A *fetch spacing* of one allows the memory interface pipeline to be full whereas a spacing equal to the fetch latency allows at most one in-flight fetch. Thus, the time required to resolve a cache miss is not deterministic but has a lower bound equal to the fetch latency. When a block is returned to the cache, the cache line is written simultaneously with the writing of the appropriate words into all registers with loads outstanding to this block. Updating all pending registers requires the multiple write ports mentioned above. This simultaneous writing is represented in Figure 3.2 by the arrows that bypass the data cache. Writing a register or a cache line is assumed to take one cycle.

3.5 Simulation Framework

This investigation of the design requirements of register files is based on execution-driven simulations using an object code instrumentation system called ATOM [Srivastava 1994]; ATOM is available for Alpha AXP workstations. This system was used to emulate the execution of an application as it would run on a virtual machine by running the application under study on an Alpha AXP workstation, the host machine. This emulation is implemented by driving a software model of the dynamically-scheduled-processor system described in the previous section with the instruction stream generated by running the application on the host machine. This instruction stream corresponds to the stream of *committed* instructions that would occur if the application were run on a dynamically-scheduled processor *with perfect branch prediction*. In other words, these instructions are those that are required to execute the application with the given input data, and are those that would be generated by a sequential processor[1].

The simulator does not model the execution of the instructions that would be incorrectly speculatively executed. This design decision was made because ATOM provides only the stream of correctly fetched and executed instructions. To build the stream of incorrectly fetched and executed instructions would have significantly complicated the simulator. To approximate the effect of fetching and executing these instructions, the simulator uses instructions from the correctly-fetched instruction stream. A branch misprediction is detected by comparing the direction predicted by the modeled branch predictor to the direction actually taken by the branch, which is supplied by ATOM. If the modeled branch predictor

[1] A sequential processor fetches the next instruction to be executed only after completing the previously issued instruction; thus, instructions are not executed in parallel.

mispredicts the direction, the software model flushes all the instructions following the branch from the (modeled) processor, and resumes execution with the instruction immediately following the branch.

By not modeling actual incorrect-path execution, the instruction-cache and data-cache behaviors might be affected. For both caches, because no incorrect-path instructions are fetched, there may be fewer conflict-induced cache misses. In addition, for the data cache, a load instruction might be issued twice with the first instance acting as a prefetch instruction if the data was returned before the branch misprediction was detected. In practice, however, since it was rare for a missing block of data to be fetched and written into the cache before the load instruction was flushed, the impact of simulating only the correct path was likely minor.

The first step in simulating the execution of an application is to compile it using the native C or Fortran compiler of the host machine. For this compilation, the global ucode optimizer was enabled and the linker was directed to perform link-time optimizations. Because the workstations used in this study were powered by the Alpha 21064 processor and because the native OSF/1 compilers could not retarget the code for a different processor, optimization and instruction scheduling assumed the use of a statically-scheduled 21064 processor. As a result, an application did not need to be recompiled for each simulation of a different virtual-machine configuration. The simulation work described in Chapters 4 to 7 is based on applications which are a subset of those in the SPEC92 and the NAS benchmark suites. A common set of benchmarks was not used since each investigation required a workload with different characteristics. For this reason, the set of benchmarks used in each chapter are discussed prior to the presentation of the results.

The simulator binary was prepared by augmenting the application binary with a procedure call before each application instruction, and linking the resulting binary with the software model. This process, illustrated in Figure 3.4, is referred to as *instrumenting* the application (binary), and was done using ATOM. When the simulator binary is run, prior to executing an instruction from the application's binary, a procedure call is made to pass to the software model information about this next-to-be-executed instruction. The software model uses the resulting instruction stream to simulate the functional behavior of the application; the memory values generated and used by the application are not simulated.

The software model simulates all steps in the execution of an instruction on a dynamically-scheduled processor. These steps include:

- Servicing of instruction cache misses;
- Fetching of instructions;

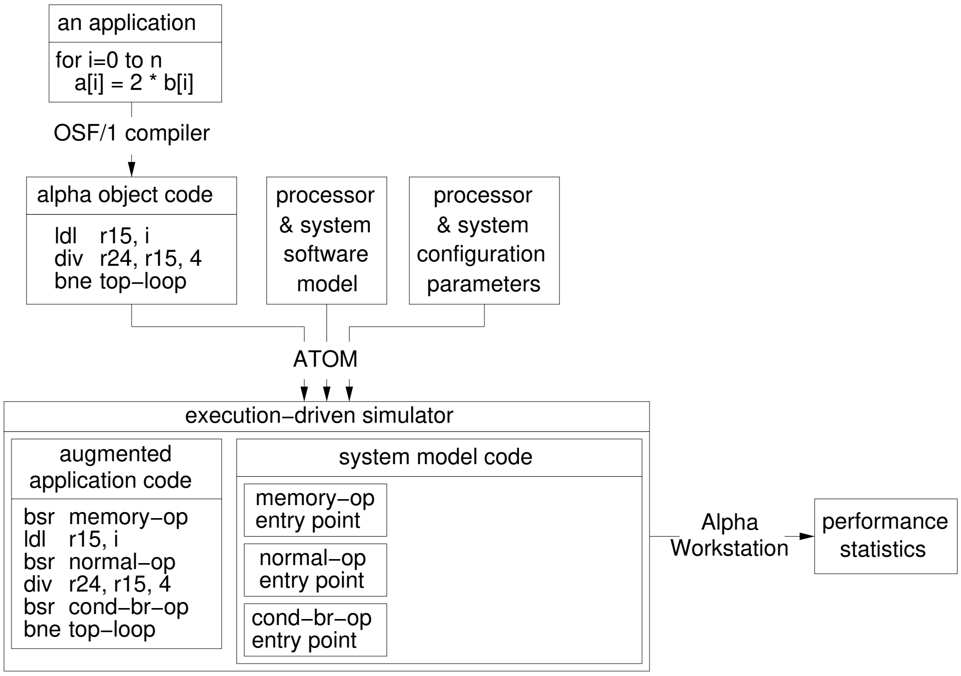


Figure 3.4: Instrumentation of an application and execution of simulator.

- Insertion of instructions into the dispatch queue;
- Allocation of physical registers;
- Selection and issuing of instructions;
- Prediction of the direction taken by conditional branches, verification of the predictions and servicing of incorrect predictions;
- Time delay in performing the indicated operation including the time required to fetch data on a cache miss;
- Completion and commitment of instructions; and
- Freeing of physical registers.

The software model also records various statistics including cache miss rates, the number of (simulated) instructions executed, and the number of (simulated) clock cycles.

When a procedure call is performed prior to the execution of an application instruction, that is, an instruction from the binary of the application, the call passes to the software model some information about the application instruction. For all instruction types, the software model is passed the instruction type and the program counter address of the instruction in the *uninstrumented* binary. Additional information is also passed with its nature dependent on the instruction type. For conditional branches, the software model is passed the name of the test register (i.e., the name of the register on which the outcome is dependent) and whether the conditional branch will be taken when it is executed. The software model considers a conditional branch mispredicted if the outcome passed to the model does not agree with the outcome predicted by the modeled branch predictor. These two branch outcomes are compared when the software model simulates the issuing of a conditional branch. Note that the register name is passed to the model but not the contents of the register because the memory behavior is not simulated. For load and store instructions, the additional information passed to the software model constitutes the number of bytes being loaded or stored (four or eight), the effective address, and the names of the required registers. Load and store instructions both require an address register, while additionally a load requires a destination register and a store requires a data register. For any other type of instruction, the additional information comprises the registers named by the instruction of which there could be one, two, or three.

Finally, a note concerning validation. Two approaches were used to validate the software model. First, to ensure that the software model correctly implemented the desired functionality, it was designed and written to identify invalid events and conditions. This error checking functionality was enabled for all of the approximately 7000 simulations that were run in the course the research described in this dissertation. Second, to determine whether the software model correctly predicted the run-time behavior of an application running on a real processor with the same functionality, the statistics obtained from these runs were compared to those reported in the literature. Discrepancies were then investigated to ensure the differences were due to differences in modeling parameters rather than modeling errors. A more direct validation of the software model was not possible because the model does not implement a functional system in terms of the actual computations. Instead, it simulates the behavior of an application with respect to how the application uses and affects processor resources, but the actual computation and output of the application are not simulated. Thus, run-time statistics and output could not be used to compare the simulated execution of an application to its execution on a real system.

This discussion concludes the examination of the system model and modeling methodology. The model is used in Chapter 5 to investigate the demands placed on the register file by the memory system for servicing data cache accesses, and is used in Chapter 4 to investigate the demands placed by other system components. Chapter 7 also uses the system model and the simulation infrastructure for the investigation of a decentralized architecture. However, for this investigation, the system model was modified to take into account the functionality required for decentralization; this functionality is described in Chapter 6.

Chapter 4

Design Implications of Non-memory-system Components

This chapter presents an investigation of the register-file design requirements for a dynamically-scheduled processor. The goal of this investigation is to provide the reader with insight into the complexity-performance relationship involving the register file and the following system aspects:

- The issue width of the processor and the number of functional units,
- The size of the dispatch queue, and
- The type of exceptions employed.

The investigation of register-file design requirements begins with a description of the benchmarks that were used. The objective of this description is to introduce important relationships and to convince the reader that the benchmarks represent reasonable workloads. Then, Section 4.2 presents an investigation of a processor with a large number of physical registers with the goal of evaluating how register requirements change as the issue width and the size of the dispatch queue are varied. From these results, a cost effective dispatch queue size is identified for the two issue widths. Then, with these dispatch queue sizes in mind, Section 4.3 investigates the register requirements and performance of the two exception models. Similarly, Section 4.4 investigates the cycle time impact of providing register files with varying degrees of functionality, under the assumption that the cycle time of the processor scales in proportion to the cycle time of the register file. The investigation is then broadened with an examination of the design requirements of various memory systems using a number of functionally balanced dispatch-queue/register-file configurations; this examination is presented in Chapter 5.

bench-mark	data set	com-mit	4-way issue							8-way issue						
			executed instr.			IPC		miss (%)		executed instr.			IPC		miss (%)	
		instr.	total	load	cbr	issue	com'it	load	cbr	total	load	cbr	issue	com'it	load	cbr
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)
compress	ref	86	126	29	14	3.06	2.09	15	14	170	42	17	4.90	2.50	10	14
doduc	small	190	209	48	12	2.75	2.49	1	10	235	56	13	4.92	3.97	1	10
espresso	ti	560	626	138	91	3.39	3.04	1	13	733	171	101	5.57	4.26	1	14
gcc1	cexp	23	27	6	3	2.80	2.35	1	19	32	8	3	4.47	3.14	1	20
mdljdp2	small	291	319	48	31	2.33	2.12	3	6	351	54	34	4.05	3.36	3	6
mdljsp2	small	350	386	82	31	2.97	2.69	1	6	429	94	33	5.25	4.28	1	6
ora	small	190	190	31	8	1.86	1.86	0	6	190	31	8	2.08	2.08	0	6
su2cor	small	417	437	107	12	3.38	3.22	17	7	460	114	13	6.24	5.65	22	7
tomcatv	ref	910	911	247	30	2.77	2.77	33	1	912	248	30	5.52	5.51	39	1

Table 4.1: Dynamic statistics for each benchmark for both issue widths. Columns 3, 4-6, and 11-13 give instruction counts in millions. Columns 9 and 16 give the data-cache miss rate for loads, while columns 10 and 17 give the conditional branch misprediction rate.

4.1 Characterization of Benchmarks

The results presented in this chapter correspond to simulations of nine of the SPEC92 benchmarks representing a balance between floating-point intensive and integer-intensive applications. The benchmarks are listed in Table 4.1 along with some run-time characteristics for the four-way and eight-way issue processors. These statistics correspond to the (simulated) execution of the nine benchmarks on a four-way issue processor and an eight-way issue processor. Both of these processors had a register file with 2048 integer registers and 2048 floating-point registers, and a 64K byte, two-way set associative lockup-free data cache with a 16-cycle fetch latency. The lockup-free cache employs an inverted MSHR organization [Farkas 1994] to process cache misses. An inverted MSHR organization can support as many in-flight cache misses as there are registers and other destinations for data in the processor. The four-way issue results are for a dispatch queue with 32 entries, while the eight-way issue results are for one with 64. The rationale behind these dispatch-queue sizes and the number of physical registers is given in the next section.

The table does not list the instruction cache miss rate because in all cases it was under 1%. Column 2 specifies which of the official SPEC92 data sets was used for the simulations. Column 3 gives the number of instructions in the trace for each benchmark obtained with the use of the noted data set. This number is equivalent to the number that commit (see Section 2.2 for the definition of *commit*). The number of committed instructions does not necessarily equal the number of instructions fetched, issued or completed due to mispredicted branches. The number of issued (i.e., executed) instructions is given in columns 4 and 11, with columns 5 and 12 giving the number of issued loads and columns 6 and 13 giving the number of issued conditional branches. Both the number of committed instructions and the number of executed instructions are dynamic instruction counts.

The average number of instructions per cycle (IPC) for each benchmark and each issue width is given in columns 7, 8, 14, and 15. The *issue IPC*, given in columns 7 and 14, is the number of instructions *issued* per (simulated) clock cycle; the issue IPC measures the rate at which instructions are dispatched to the functional units. In the system model, the difference between the issue IPC and the maximum issue width is due to the dependences in the code and the number and type of functional units. The *commit IPC*, given in columns 8 and 15, is the number of instructions that *commit* per (simulated) clock cycle. The difference between the issue IPC and the commit IPC is due to instructions that are incorrectly speculatively executed when following mispredicted branches. The commit IPC values are optimistic in part due to the use of a larger and less restricted set of functional units than is used in many recently-announced microprocessors. The commit IPC values are also high due to the assumption of a bandwidth-unconstrained memory system. To illustrate this point, consider the commit IPC value given in Table 4.1 for *tomcatv* using the eight-way issue processor. If the non-blocking cache is replaced with a blocking one, the commit IPC decreases from 5.51 to 3.86. Memory system design considerations are explored further in Chapter 5.

To put the number of committed instructions in more understandable terms, the run-time in seconds can be computed by assuming the processors ran at 200 MHz, and dividing the (simulated) clock cycle counts by this value. The clock-cycle counts are derived by dividing the number of committed instructions for each benchmark by the average commit IPC values given for each system in columns 8 and 15. The results of these computations are that on the four-way issue processor, the trace sizes correspond to between 0.1 ($=\frac{23}{200\times1.27}$) and 2.8 ($=\frac{910}{200\times1.63}$) seconds, while on the eight-way issue processor, they correspond to between 0.1 ($=\frac{23}{200\times1.45}$) and 1.9 ($=\frac{910}{200\times2.40}$) seconds. Although these run-times are not large, they are nevertheless significant.

The branch misprediction rates are given along with the overall cache miss rates for loads in columns 9, 10, 16 and 17. The misprediction rates shown are larger than those reported by McFarling [1993] for the same branch prediction scheme and a statically scheduled processor. The increase is in part due to the use of the dispatch queue in a dynamically scheduled processor. The dispatch queue increases the time between the making of the prediction and the updating of the predictor tables with the direction taken by the branch. The prediction is made at the point of insertion into the dispatch queue while the updating occurs at the point of executing the branch instruction. Hence, predictions are based on information that may not reflect the direction actually taken by immediately preceding branches in the fetch order. In addition, the information used reflects the execution order rather than the program order of branches, and these two are not necessarily the same. In practice, however, while the branch prediction accuracy did improve somewhat with in-order execution of conditional branches, this improvement occurred at the expense of a notable decrease in the commit IPC. Hence, branches are allowed to execute out of order.

4.2 How Live Registers are Used

To investigate the register file requirements under variations in the dispatch queue size and issue width, the system model was configured with 2048 integer and 2048 floating-point registers. These values were chosen to minimize the impact on the measurements from instruction-stream stalls caused by a lack of free registers. Such stalls occur if an instruction cannot be inserted into the dispatch queue because there are no free registers; in the simulations, such stalls accounted for much less than 1% of the run time. Using this system model, the number of live registers was measured in terms of the 90^{th} percentile. The 90^{th} percentile indicates how many registers should be provided by the hardware to achieve nearly the same average commit IPC as is achieved with 2048 registers[1]. The 90^{th} percentile was chosen in lieu of a geometric mean or an arithmetic average owing to the non-uniform and non-Gaussian shape of the distributions.

In examining the relationship between the dispatch queue and the number of registers, it is useful to place the registers that are live into one of four categories. The live registers in the system may be:

1. Assigned to instructions residing in the dispatch queue,
2. Assigned to instructions presently being executed (i.e., in-flight),
3. Waiting for the imprecise exception register-freeing requirements to be met, or
4. Waiting for the precise exception register-freeing requirements to be met.

Applying this categorization to the integer and floating point registers, the two issue widths yields four

[1] The 90^{th} percentile is determined by first having the simulator record how many registers were live in each cycle of a benchmark's execution. Then, this distribution of cycle counts for each register value is normalized by the (simulated) run time of the benchmark, giving normalized cycle counts of between zero and one. Next, the normalized distributions for all benchmarks of a given system model are averaged together. Finally, the number of registers needed to cover 90% of the resulting distribution is determined. This approach was adopted to prevent the distribution of a single benchmark from dominating the combined distribution.

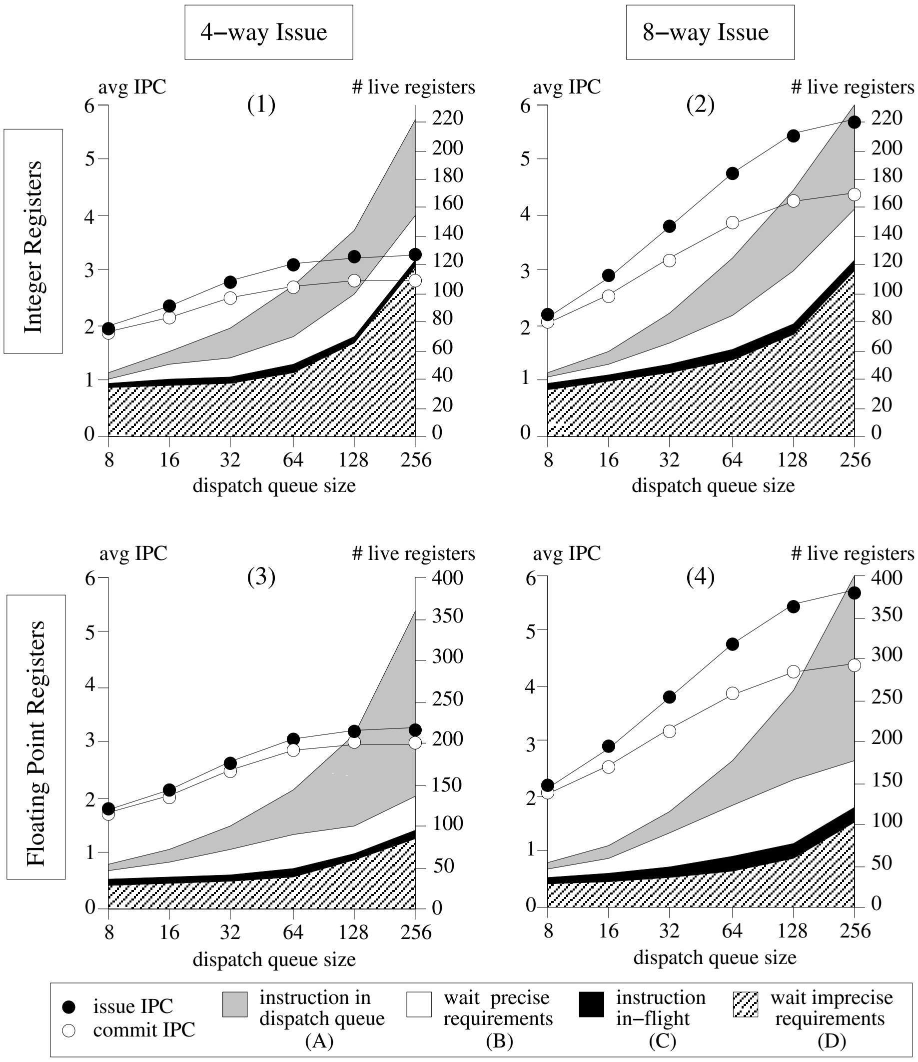


Figure 4.1: Average IPC and 90^{th} percentile number of live registers for all benchmarks as a function of the size of the dispatch queue. The shaded areas indicate the fraction of the live registers in each of four states.

sets of data[2] (Figure 4.1).

To introduce the common trends in Figure 4.1, consider first the graph corresponding to the integer registers of the four-way issue processor (graph 1). This graph presents the average issue IPC, the average commit IPC and the number of live registers as functions of the dispatch queue size using the baseline cache configuration of a 64K byte, two-way set-associative lockup-free cache with a 16-cycle fetch latency. As shown by the curve marked with the solid circles, the issue IPC approaches the issue width of the processor as the size of the dispatch queue is increased. This trend is a by-product of the

[2] In Figure 4.1 and all subsequent figures in this chapter that present averages for all benchmarks, the curves for the integer registers include data from all benchmarks whereas the floating-point register curves only include data from the floating point intensive benchmarks.

increased scheduling flexibility afforded by the larger pool of instructions available for scheduling, with the effect of allowing the scheduler to avoid many functional unit conflicts and data dependences.

The commit IPC, as shown by the curve marked with open circles, also increases with increasing dispatch queue sizes, but at a lower rate than that of the issue IPC. This difference in rates is due to the issuing of instructions that don't commit because a preceding branch in the program order was mispredicted. Also note that the gap between the issue and commit IPC is significantly larger for the eight-way issue processor than for the four-way issue processor. This phenomenon is a result of the eight-way issue processor speculatively executing more instructions, and these instructions are subject to branch misprediction.

Turning to the other curves in the graph, the upper boundary of the "instruction in the dispatch queue" region (region A) corresponds to the number of registers live under the precise exception model. The size of the "wait precise requirements" region (region B) indicates the number of additional registers required to support precise exceptions over imprecise exceptions. Note that there are at least 32 live registers. This value is the minimum number of live registers under both exception models for any program that references all architectural registers, as do most useful programs. With fewer than 32 physical registers, the system will become deadlocked because, to free a physical register, at some point there must be two physical registers assigned to the same architectural register, and there are 31 architectural registers that can be renamed (the zero register is not renamed). This arrangement is needed to cause the killing of the architectural-to-physical mapping, as discussed in Section 2.2.

The shape of the upper boundary of the A region shows that the number of live registers increases with increasing dispatch queue size. There are two primary reasons for this relationship. First, as the number of entries in the dispatch queue increases, instructions will likely be issued in an order less and less similar to the program order; in other words, there will be more out of order issue. When these instructions complete, their destination registers cannot be freed until the conditions for the exception model are met. Since these conditions involve the completion of instructions earlier in the program order, the registers will remain live longer. Hence, there will be an increase in the number of live registers waiting for precise and imprecise freeing requirements to be met. The increase under imprecise exceptions is illustrated in Figure 4.1(1) by the D region; the increase under precise exceptions is illustrated by the B region. Observe that the D region exhibits a more substantial increase in size with larger dispatch queues than does the B region. The B region represents registers assigned to instructions that have already satisfied the requirements for imprecise exceptions. Because these requirements involve the completion of all preceding conditional branches in the program order, instructions in the B category must have had any preceding conditional branches already complete. This fact reduces the disorder of instructions in the B category, and, hence, the number of registers pending completion.

Second, as the number of entries in the dispatch queue increases, more instructions remain in the dispatch queue longer. Because registers are allocated to these instructions when they are inserted into the queue, there will be an increase in the number of live registers. In addition, the slight increase in issue IPC gives rise to only a slight increase in the number of instructions in flight (the C region). This trend is also present in Figures 4.1(2), (3) and (4). Observe that the doubling of the issue width results in less than a doubling of the number of registers associated with instructions in flight. This phenomenon is due to the less than doubling in the issue IPC that occurs with a doubling in the issue width.

Finally, observe that as the dispatch queue size increases, there is a value at which the average commit IPC approaches its asymptotic value. This point occurs at a dispatch-queue size of about 32 for the four-way issue processor, and 64 for the eight-way issue processor. Moreover, once a certain number of dispatch queue entries is reached, a greater proportion of the increase in the number of live registers is attributable to the instructions residing in the dispatch queue. Taken together, these two trends suggest that these numbers of entries are respectively most cost-effective for the two processors.

4.3 Exception Model Implications

The non-zero size of the precise region in Figure 4.1 suggests that the precise exception model requires more registers than does the imprecise exception model. To examine this trend in more detail, consider first the register usage histograms of the floating point registers for *tomcatv* for both exception models. Figure 4.2 shows these histograms as run-time coverage curves. In this figure, the x-axis specifies the number of registers live at each cycle during the execution of the benchmark while the y-axis indicates the percentage of the total number of cycles with at most the indicated number of registers live. For example, on an eight-way issue processor under the precise exception model, for 70% of the run time there were 150 or fewer floating-point registers live.

Observe that the floating-point register count at which the imprecise exception model reaches 100% coverage is $\approx$130 while the same point for precise exceptions is $\approx$500. Also observe that the curve representing imprecise exceptions has shifted towards zero, an indication that fewer registers are required with this exception model. With precise exceptions, the corresponding curve exhibits a flat region between 150 and 400 registers, signifying that there were rarely 150 to 400 registers live, and that the register usage distribution is bimodal. The second modality, which is centered around 450,

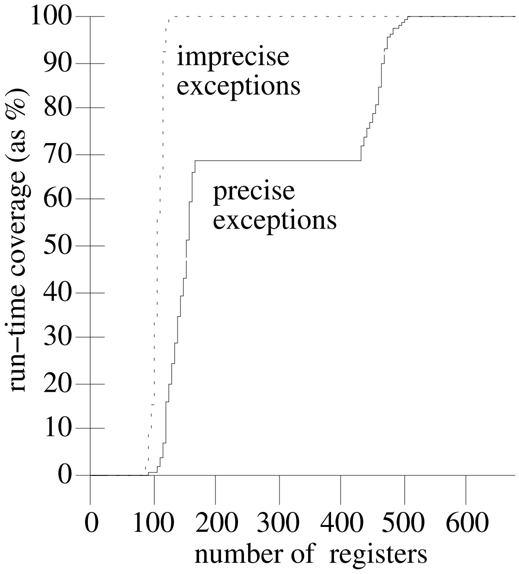


Figure 4.2: Impact of the exception model on the floating pointer registers for *tomcatv* with an eight-way issue processor, a 64 entry dispatch queue, and a lockup-free cache.

is a result of the more strict conditions for freeing registers under the precise exception model. Even though the dispatch queue has only 64 entries, the need for 500 registers in the precise model shows that at some points there is at least one instruction in the dispatch queue which is 500 instructions out of sequence (i.e., there is an instruction in the dispatch queue which occurs at least 500 instructions later in the program order than the earliest instruction). And because the 499 intervening instructions cannot be committed until the earliest instruction completes, any registers assigned to these instructions cannot be freed.

Although *tomcatv* represents an extreme case for register usage, the average 100% coverage points for all benchmarks are still significant, as shown in Figure 4.3. This figure gives the run-time coverage for both issue widths and register files; the curves were obtained by averaging the run-time coverage curves for each benchmark. As shown in the figure, 90% coverage is achieved with 90 registers for the four-way issue processor and 150 registers for the eight-way issue processor. Unfortunately, providing even 90 registers with sufficient numbers of read and write ports could be prohibitively expensive.

To evaluate the impact on performance of using a smaller and more realistic number of registers, the benchmarks were simulated with different register-file sizes while keeping the dispatch queue size constant. The results of this evaluation are presented in Figure 4.4 for both issue widths and both exception models. As shown by the solid lines in the figure, the commit IPC increases with larger register files, but the degree of improvement diminishes at the larger sizes. This trend is due to the

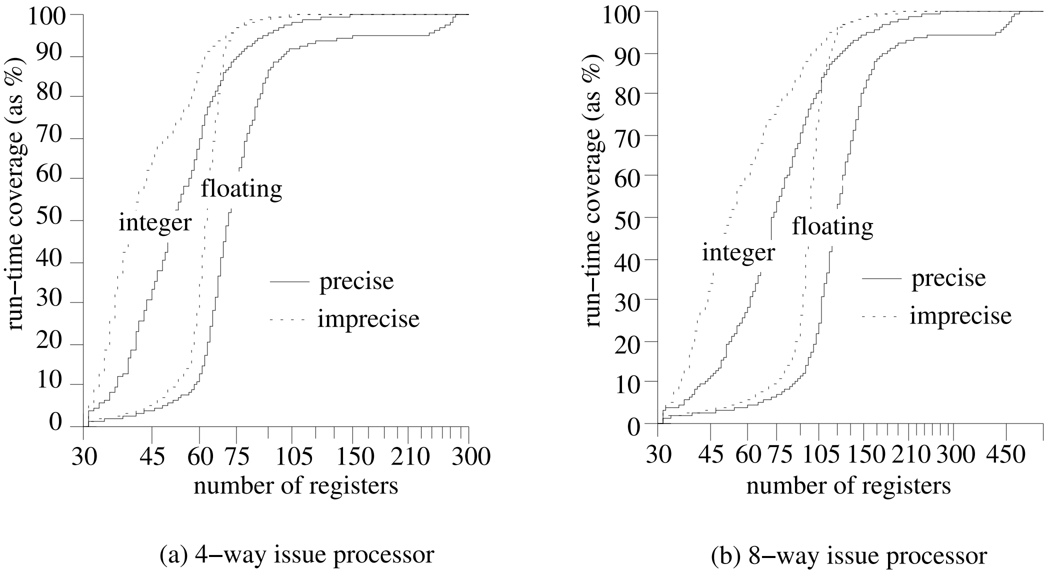


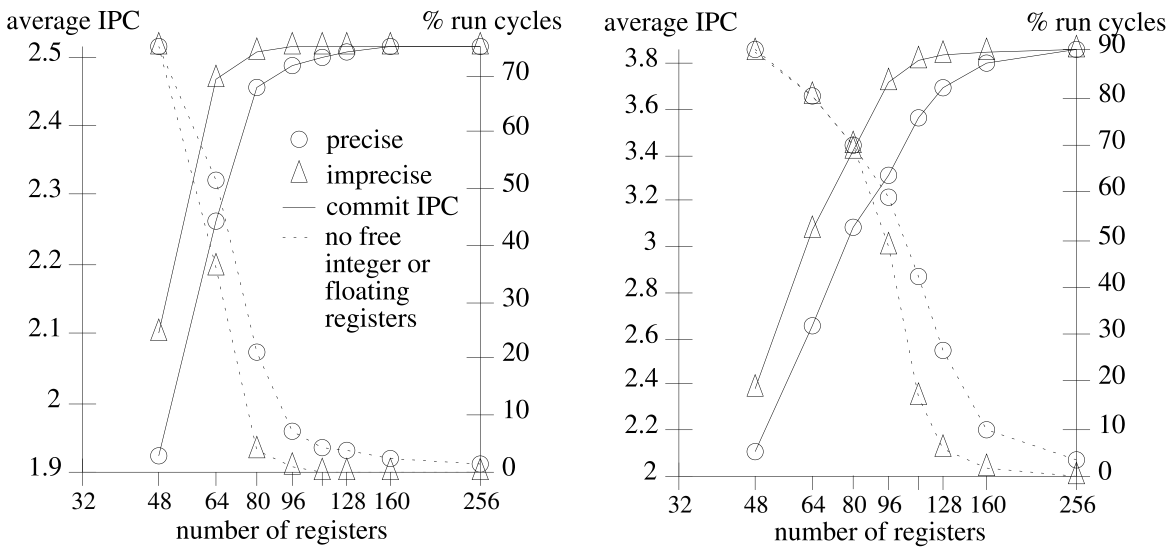
Figure 4.3: Average register usage histograms under both the precise and imprecise exception models using a lockup-free cache. The four-way issue processor histograms correspond to a dispatch queue with 32 entries while those for the eight-way issue processor correspond to a dispatch queue of 64 entries.

decreased pressure on the registers with larger register-file sizes. The register pressure is the percentage of the run-time for which there were no free registers. Observe that with larger register files, there are usually free registers available, a fact that accounts both for the leveling off of the performance, and for the similar performance under both exception models. With smaller register files, there is a more significant performance difference between the two exception models. The performance difference arises because under the imprecise model, on average, registers are live for shorter amounts of time.

4.4 Register File Timing

Thus far in this chapter, performance has been measured in terms of the average number of instructions per cycle, an approach based on the assumption that the cycle time of all processor configurations is the same. While this assumption is necessary to isolate the relationships of interest and to facilitate their understanding, variations in cycle time give rise to relationships not yet identified. This section examines these relationships using the assumption that the cycle time of the processor scales with the cycle time of the register file.

In a wide-issue dynamically scheduled processor, there are a number of critical paths that will likely determine the cycle time. These paths include the dispatch queue, the register renaming unit, and the register file. The implementation size and complexity of these structures tend to scale together since it is



(a) four-way issue processor

(b) eight-way issue processor

Figure 4.4: Average commit IPC for the benchmarks and the percentage of the run time during which there were no free registers. This percentage gives an indication of the register pressure. The dispatch queue size was held constant and the number of registers varied.

desirable that no one of these structures offer a disproportionate amount of functionality. For example, as illustrated by the results presented in this chapter, if many additional entries are added to the dispatch queue, additional registers in the register file will probably be needed as well.

The register renaming unit and the dispatch queue are subject to wide variations in implementation architecture and circuitry, while the register file design space is more limited. However, independent of how the dispatch queue and renaming unit are implemented, they will have structures similar to those found in the register file (such as numbers of ports or numbers of entries). Therefore, in terms of cycle time, the register file, the register renaming unit, and the dispatch queue are assumed to scale similarly to each other, and therefore to the system as a whole.

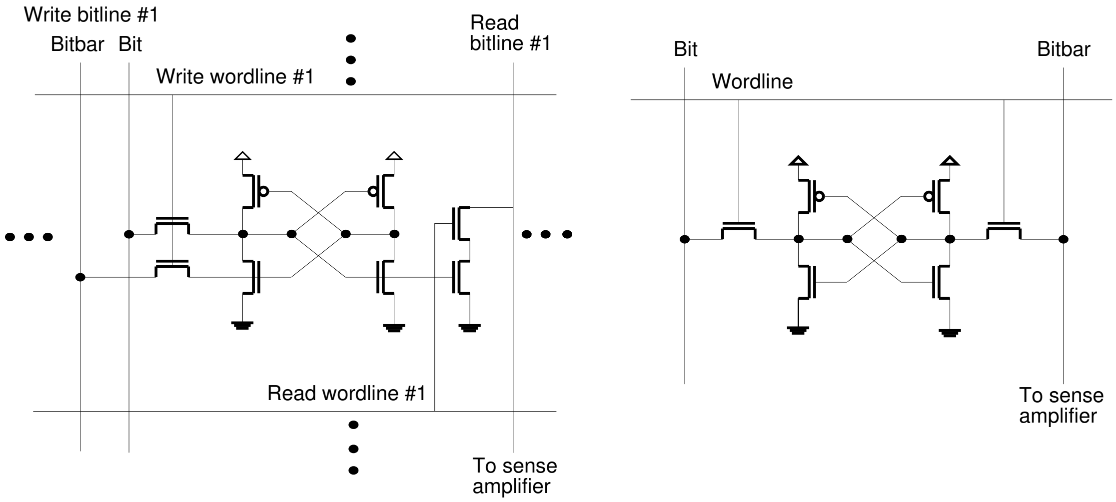
An evaluation is presented below of the register file cycle times required for the four-way and eight-way issue processor systems. This evaluation assumed a lockup-free cache organization and a 32-entry dispatch queue for the four-way issue processor, and a 64-entry dispatch queue for the eight-way issue processor. Various register file designs were simulated each differing in the number of read and write ports and the number of registers. The number of read and write ports was set by the issue width of the processor, while the register file sizes correspond to those used in Figure 4.4. For the four-way issue processor, the integer register file was assumed to have eight read ports and four write ports, whereas the floating-point register file had half as many (because only half as many floating-point instructions can be issued per cycle in the processor model). The eight-way issue processor was assumed to include double the number of ports assumed for the four-way issue processor.

In these simulations, the cycle times of the multiported register files were generated using the register file cycle time (RFCT) model [Jouppi] that is based on the *CACTI* cache access and cycle time model of Wilton and Jouppi [Wilton 1994]. The CACTI model, consisting of relatively simple equations, can be used to predict the read access and cycle times of caches as a function of various cache, process, and array-organization parameters. To evaluate the cycle time of register files, part of the CACTI model was disabled and various parameters were changed. The following summarizes the changes necessary for these evaluations.

4.4.1 The Register File Timing Model

Unlike a cache, a register file does not require a tag array because, in a register file, there is a one-to-one correspondence between register IDs and memory locations. Thus, the tag array of the CACTI model is disabled and only the data array is used. The RFCT model assumes a data array organization with no two registers sharing a wordline. This organization prevents the occurrence of stalls induced by the simultaneous access of two registers that share a wordline. Such stalls are more acceptable in caches which have larger memory arrays and therefore have a smaller probability for conflicts. Thus, the assumed array organization of the register file is an array with as many columns as bits in each register, and as many rows as there are registers in the register file.

Caches are not usually required to be simultaneously read and written, in part because of the smaller probability of needing such functionality, and in part because such functionality has a significant area cost. Register files, however, must be simultaneously read and written, and, to support this functionality, separate read and write bitlines are required. Figure 4.5(a) depicts the register file memory cell with these bitlines, whereas Figure 4.5(b) depicts the cache memory cell with one less bitline. Note that unlike the cache cell, the register file cell uses a single bitline for reading, rather two bitlines. Single-ended reading requires one fewer bitline per read port, and was assumed to reduce the dimensions of a memory cell. Also note that in the register file cell, the read-access transistors do not directly connect the bitlines to the internal nodes. Rather, a two-transistor NAND stack is used to connect each bitline to the internal nodes of a cell. This NAND stack is necessary owing to the large number of ports, and the likelihood that a cell may be read concurrently by multiple ports. Such concurrent reading could disturb the cell



(a) Multiported register file cell

(b) Cache cell

Figure 4.5: Memory cells for the register file and cache-access timing models.

and inadvertently cause it to be written if pass transistors were used to provide read access.

For the register-file cell, as in the case of the cache cell, writes are performed differentially. Since the bitlines for writes are driven by external, and potentially very powerful, driver circuits, writes are not assumed to be on the critical path. Thus, the cycle time of the register file is assumed to be governed by the read time.

Cycle Time of a Cache

A quantitative expression for the cycle time of register file was derived from the cycle time discussion presented by Wilton and Jouppi [Wilton 1994]. Quantitatively, their results were expressed in their equations 38 and 40, which are reproduced below as equations 4.1 and 4.2.

$$
\begin{aligned}
T_{access,dm} &= \max(T_{dataside} + T_{outdrive,data},\ T_{tagside,dm}) && (4.1)\\
T_{dataside} &= T_{decoder,data} + T_{wordline,data} + T_{bitline,data} + T_{sense,data} &&\\
T_{tagside,dm} &= T_{decoder,tag} + T_{wordline,tag} + T_{bitline,tag} + T_{sense,tag} + T_{compare} + T_{valid} &&\\
T_{cycle} &= T_{access} + T_{precharge} && (4.2)
\end{aligned}
$$

In equation 4.1, $T_{access,dm}$ is the access time of a direct-mapped cache. The access time is the time after the address inputs are valid until the request data is driven from the cache. It is equal to the larger of the timing path through the data array ($T_{dataside}$) or the timing path through the tag array ($T_{tagside,dm}$). The remaining terms of equation 4.1 are defined as follows:

- $T_{decoder,data}$ ($T_{decoder,tag}$) is the time required to decode the cache line that is being read in the data (tag) array and select the wordline in the data (tag) array.
- $T_{wordline,data}$ ($T_{wordline,tag}$) is the time required to drive the selected wordline in the data (tag) array.
- $T_{bitline,data}$ ($T_{bitline,tag}$) is the time delay between the data (tag) wordline going high and the sense amplified being able to detect the state of the cell.
- $T_{sense,data}$ ($T_{sense,tag}$) is the time delay through the sense amplifier of the data (tag) array.
- $T_{outdrive,data}$ is the time required to drive the read data onto the data bus.
- $T_{compare}$ is the time required to compare the tag read from the tag array to the (address) tag of the data for which the cache probe is being done.
- T_{valid} is the time required to drive the valid signal onto the data bus.

The cycle time of the cache, which is the minimum time between the start of consecutive accesses, is the access time (equation 4.1) plus the time to precharge the bitlines, comparators, and internal decoder bus ($T_{precharge}$). Equation 4.2 gives the cycle time of a direct-mapped cache.

Cycle Time of the Register File

The cycle time of the register file is derived by disregarding all terms that correspond to the tag array in equations 4.1 and 4.2. The tag array, as noted above, is unnecessary because in a register file, there is a one-to-one correspondence between register IDs and memory locations. Using these two equations, the cycle time of the register file can be written as:

$$T_{cycle,reg} = T_{decoder} + T_{wordline} + T_{bitline} + T_{sense} + T_{outdrive} + T_{precharge} \tag{4.3}$$

For the RFCT model, it was assumed that the single-ended sensing of the register file bitlines would have comparable timing to the differential sensing driven off internal memory cell nodes in the CACTI model. The timing differences caused by the increased number of wordlines and bitlines were taken into account by scaling the capacitances for the metal used to wire up the wordlines and bitlines. The scaling was directly proportional to the number of bitlines and wordlines, and, hence, to the number of ports of the register files considered. Three different register files were considered:

1. A register file with 6 ports (4 for reading, 2 for writing) which is used for the 4-way issue processor's floating point register file,

2. A register file with 12 ports (8 for reading, 4 for writing), which is used for both the 4-way issue processor's integer register file and the 8-way issue processor's floating point register file, and

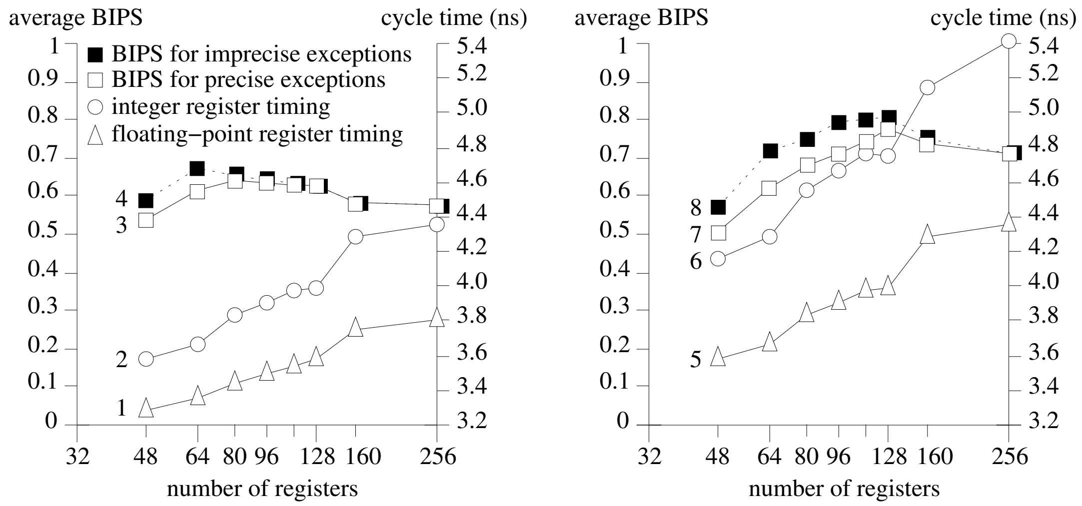
3. A register file with 24 ports (16 for reading, 8 for writing), which is used for the 8-way issue processor's integer register file.

To illustrate the scaling, consider the 6 port register file. For this register file, the number of bitlines per memory cell is 8 (= 4 read ports $\times$ 1 $\frac{bitline}{read\ port}$ + 2 write ports $\times$ 2 $\frac{bitlines}{write\ port}$). In comparison, the cache memory cell requires only 2 bitlines. Thus, the capacitance in the wordline direction for the CACTI model was scaled by 4 (= $\frac{8}{2}$). For the bitlines, their length and hence capacitance increases with increasing numbers of wordlines. The 6 port register file requires 6 wordlines compared to the 1 required for the CACTI memory cell. Since each wordline is assumed to account for 25% of a cell's height in the CACTI memory cell, each additional wordline in the RFCT mode is assumed to add $0.25 \times C_{bitmetal}$ to the capacitance in the bitline direction, where $C_{bitmetal}$ is the wordline contribution to the capacitance of a bitline in the CACTI model. A 0.5μm CMOS technology was assumed for determining the value of the technology parameters.

4.4.2 Cycle Time Evaluation

Using the register file cycle time model, the cycle time was determined for each of the integer and floating-point register files as a function of the size of the register file. The results of the evaluation are presented graphically in Figure 4.6. This figure presents for both issue widths two register file timing curves and two estimated performance curves.

In Figure 4.6, curves 1 and 5 indicate the cycle time of the floating-point register file, while curves 2 and 6 indicate the cycle time of the integer register file. Note that the cycle time of the floating point register file is always below that of the integer register file. The speed difference is attributable to the floating-point register file having half as many ports as the integer register file. Note also that if the



(a) Four−way issue processor

(b) Eight−way issue processor

Figure 4.6: Register file timing and estimated system performance. The four-way issue processor used a 32 entry dispatch queue while the eight-way issue processor used a 64 entry dispatch queue.

number of registers is doubled for the four-way issue processor, the increase in the cycle time of the register file is smaller than the increase that occurs if the number of registers is kept constant and the eight-way issue processor is used instead. This relationship is due to the cycle time of a large register file being more strongly affected by the number of register file ports than by the number of registers. For a register file, doubling the number of ports doubles the number of wordlines and bitlines (quadrupling the register file area in the limit), but doubling the number of registers only doubles the number of wordlines (doubling the register file size in the limit).

Figure 4.6 also presents an estimate of system performance assuming that the system cycle time scales proportionally to that of the integer register file. Performance is measured in billions of instructions per second (BIPS) and is derived by dividing the average commit IPC from Figure 4.4 (page 70) by the cycle time of the register file in question. The performance obtained under precise exceptions is indicated by curves 3 and 7 while that obtained under imprecise exceptions is indicated by the curves 4 and 8. For both issue widths, the imprecise model has a small performance advantage with small numbers of registers. However, in the four-way issue processor, there is little performance difference between the two exception models with register file sizes greater than 80. For the eight-way issue processor, this point occurs at 160 registers, a result of the need for more registers due to the greater incidence of out-of-order issue of instructions in the eight-way issue processor.

The performance curves in Figure 4.6 all exhibit performance maxima at moderate numbers of registers. Below the optimum number of registers, the average BIPS falls off, a result of instruction-stream stalls. Above the optimum number of registers, the increasing register file cycle time negatively impacts the system cycle time and, hence, overall performance. Comparison of the performance maxima for each of the two issue widths indicates that the maximum performance of the eight-way issue processor is only 20% better than that of four-issue processor. A major reason for this fact is the large increase in the cycle time that is mandated by the larger and more complex register file.

4.5 Chapter Summary

This chapter has described the investigation of a number of issues in the design of register files for dynamically-scheduled processors. Several observations can be made.

First, doubling the size of the dispatch queue resulted in less than a doubling in the commit IPC. Moreover, once a certain number of dispatch-queue entries was reached, a greater proportion of the increase in the number of live registers was attributable to the instructions residing in the dispatch queue. Taken together, these two trends suggest that there exists a small number of dispatch-queue sizes that are most cost-effective for the modeled processors.

Second, additional registers were required with precise exceptions in the modeled processors to maintain performance comparable to that obtained with imprecise exceptions. The imprecise model that was simulated only reduced the average number of registers required by the four-way issue machine by at most 20% with a dispatch queue of 32 entries. The difference between the precise and imprecise models was larger for the eight-way issue machine, since, to get good utilization of the eight-way issue machine, the instructions must be executed in out-of-fetch order to a greater extent. The eight-way issue machine using imprecise exceptions required an average of 37% fewer registers than one using precise exceptions with a dispatch queue size of 64. Because the register file cycle time is more heavily dependent on the number of register file ports than on the number of registers, and in view of all the other hardware required by a dynamically scheduled superscalar processor, the cost of the additional registers required to support precise exceptions is relatively small.

Third, the processor at times could use many hundreds of registers. However, when the number of registers was limited to 80 (both integer and floating-point) for the four-way issue machine and to 128 for the eight-way issue machine, the performance (i.e., clock-cycle count) was only a few percent lower than that of a machine with an unlimited number of registers. The number of registers used by an application is also affected by the design of the memory system, a subject examined in detail in the next chapter.

Fourth, there are many critical paths in a dynamically scheduled superscalar processor; the worst may have timing that scales similarly to that of register files since the implementation size and the complexity of components tend to scale together. Therefore, the scaling of machine cycle time with complexity was approximated as being proportional to the scaling of the required register file cycle time. Since the register file becomes slower as the number of registers increases, and the resulting IPC tends to saturate, the overall machine performance has a maximum with the above noted number of registers. In addition, because the register file cycle time is also strongly dependent on the number of ports, the simulation results suggest that the use of centralized integer and floating-point register files may yield only a 20% performance improvement for an eight-way issue processor over four-way issue machine. This result suggests the following hypothesis. Achieving larger performance gains may only be possible by simplifying the components that impose limits on the machine's cycle time by partitioning them. This hypothesis is one of the motivations for *decentralized architectures*. A novel decentralized architecture is examined in Chapters 6 and 7.

This discussion concludes the investigation of the demands placed on the design of the register file by non-memory-system components. The investigation of the design requirements of the register file is continued in the next chapter in terms of the demands placed by the memory system components used to service requests for data.

Chapter 5

Design Implications of Memory-System Components

This chapter presents the results of an investigation of the performance relationships involving the components in the data memory hierarchy. The goal of this investigation is to provide insight into the effectiveness of hardware-based techniques for reducing the apparent time-cost of cache misses or increasing the tolerance for data-cache misses. The components that are considered are the register file, the lockup-free data cache (i.e., miss status holding registers and the address stack), and the hardware prefetcher (i.e., stream buffers). The interface between these components and the lower levels of the memory hierarchy is also examined. A block diagram of these components is given in Figure 5.1 along with several of the design parameters assumed in the investigation. These parameters are described in subsequent sections. The remainder of this introductory section reviews the operation of these components, discusses their impact on the cache-miss latency and the tolerance for cache misses, and describes how they are incorporated into the system model.

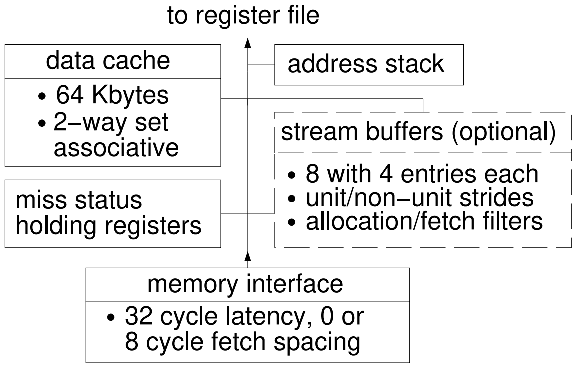


Figure 5.1: Block diagram of the memory-system components used for servicing data-cache misses.

Dynamically-scheduled processors hide the time penalty of servicing a cache miss by issuing instructions from later in the fetch stream while the cache miss is being resolved and before the instructions that are dependent on the cache miss are issued. The ability to tolerate a cache miss is a balance between the time required to resolve the miss and the number of instructions available that can be issued because they have no outstanding dependences. This balance exists because the longer a cache miss takes to be resolved, the greater the number of instructions that are required to hide it. The number of such instructions depends on the ability of the hardware to fetch new instructions and the availability of the hardware resources required to process them. To reduce the performance impact of cache misses, either the time required to resolve them can be reduced, or the supply of ready-to-issue instructions can be increased.

Two approaches for reducing the cache-miss latency are to increase the bandwidth of the memory interface, and to include stream buffers in a system. A memory interface with fetch latency l and limited bandwidth may be modeled as a pipeline with l stages into which requests can be launched every n cycles, where n is the fetch spacing. If $n = 0$, requests are launched as soon as they are submitted to the interface and the effective cache-miss latency will be deterministic, constant, and equal to l. However, if $n > 0$, a request R cannot be issued until n cycles after all earlier-submitted requests have been issued. Because the time R waits is a function of the temporal relationship between it and the previously submitted fetch requests, the effective cache-miss latency will be variable, probably larger than l, and likely non-deterministic.

Stream buffers can be used to reduce the effective cache-miss latency, but a reduction is not guaranteed. Stream buffers seek to reduce the effective cache-miss latency by prefetching data in the hope that it will be required to resolve a subsequently detected cache miss. Although the cache-block-fetch requests initiated by the stream buffers are only launched into the lower levels of the memory hierarchy when there are no current data-cache requests, in systems with a non-zero fetch spacing n, a previously-launched stream-buffer request might delay the launching of data-cache-fetch requests by up to $n - 1$ cycles. If these delays occur sufficiently often, the effective cache-miss latency may increase rather than decrease, which would worsen performance.

The supply of ready-to-issue instructions is affected by the branch predictor, the resources for processing misses, the dispatch queue, and the register file. The branch predictor is important because it enables the hardware to continue fetching instructions after encountering a branch in the fetch stream for which either the branch direction or the branch destination (address) is not statically known. In the system model used for this dissertation, only the direction of conditional branches is predicted; the direction and destination of all other branches are assumed to be 100% predictable.

The miss-processing resources are also important; these resources are the MSHRs and the address stack. As explained in Section 2.1.5, when a cache miss is detected, the MSHRs are associatively searched to determine whether the missing cache block is already being fetched. If it is not, then an MSHR is allocated, and a fetch request is submitted. The address stack is used to resolve cache misses once a cache block has been returned, either by the memory interface, or by a stream buffer if stream buffers exist in the system. The number of address-stack entries and MSHRs has an impact because, if a cache miss occurs and either resource is not available, the hardware must stop issuing memory instructions until the required resource becomes available. Moreover, the hardware must not issue the instructions that are dependent on the memory instructions. As the stall progresses, the hardware will continue inserting, when possible, new instructions into the dispatch queue. But, because it is likely these are either memory instructions or instructions that are dependent on the instructions already in the queue, the queue will tend to fill up with instructions that cannot be issued. Thus, as the stall progresses, the rate of inserting new instructions will tend to decrease, and the tolerance for cache misses will decline. This decline can be offset by increasing the number of dispatch-queue entries and physical registers.

The number of physical registers is important because the architectural registers named by instructions must be mapped to physical registers when the instructions are inserted into the dispatch queue. If a register is not available and an instruction requires one, the hardware must stop inserting new instructions. The availability of registers is affected in a number of ways by the cache miss rate, the effective cache-miss latency, and the amount of support for in-flight misses. First, if the effective cache-miss latency increases, the target registers of loads that incur a cache miss will have to remain live longer. Since the hardware may have to delay the issuing of instructions that use the results of these loads, the length of time that their source and destination registers are live may also increase. Second, if an address-stack entry or an MSHR is required in the processing of a miss and either is not available, a stall in the issuing of memory instructions is required. Stalls caused by these structural hazards may increase the length of time that the source and destination registers of unissued memory instructions are live. Finally, when the issuing of instructions is delayed, more instruction-issue disorder is likely, which increases the time required to meet the conditions imposed by the exception model for freeing registers.

This chapter presents the results of an investigation concerning the effectiveness of the above hardware-based techniques for reducing the apparent time-cost of cache misses and increasing the tolerance for data-cache misses. The discussion begins with a description of the benchmarks used in this study. Then, Section 5.2 presents an examination of different implementations of lockup-free data caches, followed by the examination in Section 5.3 of the implications of including stream buffers. Finally, Section 5.4 presents some concluding remarks.

5.1 Benchmark Characterization

This section describes some of the characteristics of the simulation workload used in this chapter (Section 3.5 describes the simulation infrastructure). The goal of the discussion is to convince the reader that the workload is appropriate for investigating memory-system implementations.

The workload consists of seven benchmarks, six from the SPEC92 suite plus the *appsp* benchmark from the NAS suite. The input data set for each benchmark was chosen with three objectives in mind:

1. The (simulated) execution of the benchmark should generate an instruction trace of reasonable length;

2. Enough data-cache behavior is generated to affect the performance of the memory-system implementations; and

3. The length of time needed to run each simulation on a DEC 3000 model 600 AXP Workstation[1] should not exceed 24 hours.

For each of the six SPEC92 benchmarks, one of the official data sets was used (ref or small), and these are shown in Table 5.1 in column 2. However, to reduce the time required to simulate *hydro2d*, the number of iterations specified in the data set was reduced, as noted in the caption of the table. The caption also describes the data set used for the NAS benchmark.

The results in Table 5.1 are for two systems, one with a four-way and the other, with an eight-way issue processor. Both systems enforced precise exceptions, and had a 64K byte, two-way set associative lockup-free data cache with no in-flight miss restrictions, a 32-cycle fetch latency, and an eight-cycle fetch spacing. The four-way issue processor had a 32-entry dispatch queue, and 64 registers (i.e., 64 integer and 64 floating point), while the eight-way issue processor had a 64-entry dispatch queue and 96 registers.

[1] This workstation has a DECchip 21064 microprocessor running at 175 MHz, separate 8K byte instruction and data caches, a 2M byte on-board secondary cache, and 128M bytes of main memory.

bench-mark	data set	com-mit instr.	4-way issue						8-way issue					
			issue instr.		IPC		% load miss		issue instr.		IPC		% load miss	
			total	load	issue	commit	pri.	sec.	total	load	issue	commit	pri.	sec.
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)
appsp	note[a]	319	320	68	1.58	1.57	6.0	6.8	322	68	2.44	2.42	6.0	7.6
compress	ref	86	111	25	1.63	1.27	11.9	3.5	139	31	2.34	1.45	10.7	3.1
hydro2d	note[b]	237	238	54	1.45	1.44	12.6	21.1	240	55	2.52	2.49	12.5	36.4
mdljdp2	small	291	317	47	1.74	1.60	2.3	0.4	352	53	2.84	2.34	2.1	1.9
su2cor	small	417	432	106	1.88	1.81	8.4	8.1	445	109	2.63	2.46	8.2	14.3
swm256	ref	377	378	97	1.77	1.76	6.6	2.6	379	97	2.92	2.92	6.6	10.3
tomcatv	ref	910	911	247	1.63	1.63	11.2	21.1	911	248	2.40	2.40	11.2	28.4

Table 5.1: Dynamic statistics for each benchmark for both issue widths. Columns 3, 4, 5, 10, and 11 give instruction counts in millions, with columns 8, 9, 14, and 15 give the primary and secondary data-cache miss rates for load instructions. Notes: (a) *appsp* was run for 50 iterations with a 12x12x12 grid; (b) *hydro2d* was run for 15 iterations rather than the 400 specified in the official "ref" data set.

The remaining columns of the table present statistics that show the first two objectives were met. The number of instructions in a trace is equivalent to the number of instructions that were committed when the benchmark was run. This number is given for each benchmark in column 3. To put these numbers in more understandable terms, the run-time in seconds can be computed by assuming the processors ran at 200 MHz, and dividing the (simulated) clock cycle counts by this value. The clock-cycle counts are derived by dividing the number of committed instructions for each benchmark by the average commit IPC values given for each system in columns 7 and 13. The results of these computations are that on the four-way issue processor, the trace sizes correspond to between 0.3 ($=\frac{86}{200\times1.27}$) and 2.8 ($=\frac{910}{200\times1.63}$) seconds, while on the eight-way issue processor, they correspond to between 0.3 ($=\frac{86}{200\times1.45}$) and 1.9 ($=\frac{910}{200\times2.40}$) seconds. Although these run-times are not large, they are nevertheless significant. To observe the amount of data-cache behavior, consider the number of issued instructions given in columns 4 and 10 and the number of issued loads given in columns 5 and 11. The data shows that loads comprise a significant proportion of the issued instructions. In addition, columns 8, 9, 14 and 15 show that a significant proportion of the issued loads result in either a primary miss (columns 8 and 14) or a secondary miss (columns 9 and 15). Since the number of issued loads and number of cache misses are significant, the second objective is satisfied.

5.2 Implications of Support for In-flight Cache Misses

An important factor that affects the performance of a dynamically-scheduled processor and the requirements of the register file is the number of in-flight misses that the lockup-free data cache can support. To evaluate the design and performance implications of such data caches, the system model was modified

processor parameters	
issue width	number registers
4-way issue with 32-entry dispatch queue	48, 64, 96, or 128
8-way issue with 64-entry dispatch queue	64, 96, 128, or 256

memory parameters			
cache parameters			fetch spacing
type	abbrev.	in-flight restrictions	
lockup	lk	not applicable	
lockup-free	m_x	at most x in-flight misses	0 or 8
	f_x	at most x in-flight fetches	
	i	none	
perfect	p	not applicable	

Table 5.2: System designs for investigation of support for in-flight misses. A system is defined by selecting a design parameter from each column.

so that the number and type of in-flight load-induced misses[2] could be restricted. This modified system model was then used to evaluate the performance of the benchmarks for a number of system designs. The design space is summarized in Table 5.2. A system is defined by selecting a design parameter from each column to determine the issue width and a register-file size for the processor, a cache design and, if applicable, a fetch spacing. For example, one of the investigated designs is a four-way issue processor with a 32-entry dispatch queue, 48 registers (i.e., 48 integer and 48 floating-point registers), and a *perfect cache*, that is, a cache with a 100% hit rate. Two other designs are defined by replacing the perfect cache with either a lockup cache or a lockup-free cache both of which are 64K byte, two-way set-associative and connect to a memory interface with a 32-cycle fetch latency. All systems assume the processor enforces precise exceptions.

For the lockup-free cache, three types of in-flight miss restrictions are considered. The most restrictive type, designated m_x, limits the number of outstanding cache misses to be at most x; x is an integer, greater than zero. The second type, designated f_x is less restrictive because it imposes no limit on the number of secondary misses. Rather, it limits only the number of primary misses to be at most x. A cache with this functionality offers greater flexibility than one with the functionality offered by m_x, and therefore has a more aggressive implementation. Because a fetch request is required for each primary miss, the value of x for the f_x restriction also indicates the maximum number of outstanding cache-initiated fetch requests. The third type, designated i, is even less restrictive because it does not impose a limit on the number of primary or secondary misses.

While the i restriction imposes no limit on the number of primary and secondary misses, and the

[2]As discussed in Section 3.4.3, store misses are implemented using write-around and write-through policies, and thus no cache-block fetch is required to resolve them.

f_x restriction imposes no limit on the number of secondary misses, a limit nonetheless exists. For the i restriction, this limit is set by the number of destinations in the processor for fetch data. In a real processor, the destinations for fetch data could include write-buffer entries (for merging with write data when writing into a write-allocate cache), the program counter, an instruction prefetch buffer (if it exists), and the integer and floating-point physical registers. However, in the system model used in this chapter the only destination for fetch data is the physical registers. Thus, if the system being modeled has R integer and R floating-point registers, then, under the i restriction, there can be as many as $2R$ in-flight misses. For the f_x restriction, on the other hand, there can only be as many as $2R - 1$ secondary misses because at least one register must be bound to the load that induced the primary miss. In practice, however, real processors do not support so many secondary misses. Rather, a processor with N_a address-stack entries and N_m MSHRs can support as many as N_a misses, N_m of which are primary misses, and $N_a - N_m$ of which are secondary misses.

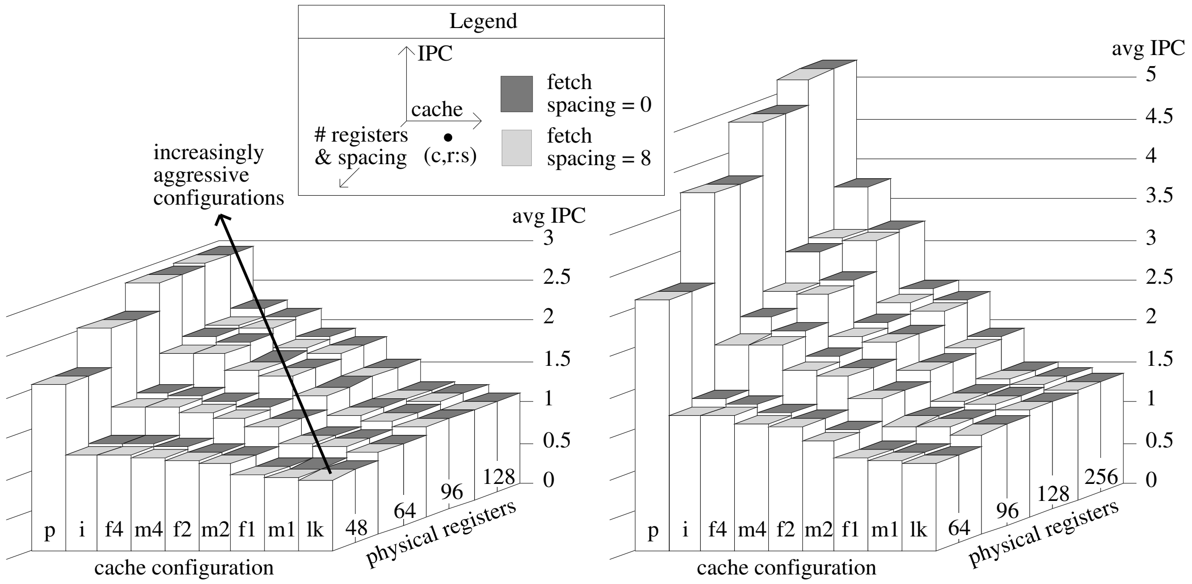
For dynamically-scheduled processors in general, while the in-flight miss restriction specifies the maximum number of in-flight primary misses, it is possible for there to be more than this number of fetches in the pipeline of the memory interface. This situation may occur when some of the in-flight fetches were initiated by load instructions that were subsequently flushed from the system before the requested data was returned during the recovery from a mispredicted branch. The definitions for f_x and m_x must be refined to take into account the difference. The maximum number of in-flight fetches is determined by the fetch spacing. With the assumed 32-cycle fetch latency and a non-zero fetch spacing, say, eight cycles, there can be at most $\frac{fetch\ latency}{fetch\ spacing} = \frac{32}{8} = 4$ in-flight fetches, but, with a zero-cycle fetch spacing, the number of in-flight fetches is bounded by the product of the fetch latency and the number of load instructions that can be issued per cycle. The difference between the maximum number of in-flight fetches and the maximum number of primary misses is due to mispredicted branches.

When a mispredicted branch is processed, all load instructions that follow the branch in fetch order are flushed from the system. If any of these loads, however, are waiting for a cache-block fetch to complete, and if the fetch is already in the pipeline of the memory interface, the fetch cannot be cancelled. Thus, the pipeline may contain requests for cache blocks that will have no corresponding load instruction when the cache blocks are returned. Moreover, there may also be no matching MSHR if MSHRs are invalidated whenever all their associated pending loads are flushed from the system during the processing of a mispredicted branch. While this invalidation policy may result in the hardware launching a fetch-request for a cache-block that is already being fetched, it reduces the possibility of a structural-hazard-induced stall occurring when processing resumes after recovery from the mispredicted branch. In the investigation described in this chapter, the processor is assumed to implement this invalidation policy. Thus, the f_x and m_x restrictions specify the maximum number of in-flight fetches for which an MSHR is assigned.

Using the system model, the performance of the benchmarks was determined on the systems listed in Table 5.2. Figure 5.2(a) and (b) respectively present the (overall) average commit IPC[3] obtained by the benchmarks on the systems with the four-way and eight-way issue processors. In both of these figures, the height of the bar at coordinates $(c, r : s)$ gives the average commit IPC for a processor having a cache design c, a register-file size r and a fetch spacing s. The tables below each graph give the explicit average commit IPC values. The cache designs are mapped on the left-to-right axis with the less restrictive designs located to the left. The register-file sizes and fetch spacings are mapped on the front-to-back axis, with the larger register files located towards the back. For each register-file size, the IPC values for a zero-cycle fetch spacing are behind those for an eight-cycle fetch spacing. Thus, the coordinates to the left and towards the back represent more aggressive system implementations.

Examination of the data presented in Figure 5.2 suggests a number of important relationships. These relationships are discussed below beginning with the relationship between performance, number of registers, and support for in-flight misses. Consider the bars in Figure 5.2(a) corresponding to the use of a lockup-free cache with no restrictions on in-flight misses, that is, the bars with coordinates $(c = \text{i}, r : s)$. Observe that, as noted in the discussion of Chapter 4, the average commit IPC increases with the size of the register file. Furthermore, the rate of increase decreases at larger sizes. For instance, for a fetch spacing of eight cycles, doubling the size from 48 to 96 yields an improvement of 70% (=$\frac{2.02}{1.19} - 1$), while doubling the size from 64 to 128 yields an improvement of only 36% (=$\frac{2.15}{1.58} - 1$). As explained in Chapter 4, in general, an increase in the number of registers permits more instructions to be in some stage of execution, thereby better utilizing the available hardware, and thus improving performance. However, this correlation between more registers and better performance is less pronounced for the systems with less support for in-flight misses. For example, if four in-flight fetches are permitted, $(c = f_4, r : s)$, doubling the size from 48 to 96 yields an improvement of 68% (=$\frac{2.01}{1.19} - 1$) while doubling the size from 64 to 128 yields an improvement of 35% (=$\frac{2.14}{1.58} - 1$). But, if the support for misses is reduced by allowing only four in-flight misses, $(c = m_4, r : s)$, the improvement percentages are 53% (=$\frac{1.80}{1.17} - 1$) and 25% (=$\frac{1.88}{1.50} - 1$), respectively.

[3] The (overall) average commit IPC is calculated for each system configuration by first computing the average commit IPC for each benchmark for that system. The per-benchmark average commit IPC is equal to the total number of instructions that are committed when the benchmark is run divided by the total number of (simulated) clock cycles required to run it. Then, these per-benchmark averages are combined using an arithmetic average to obtain the reported (overall) average commit IPC.



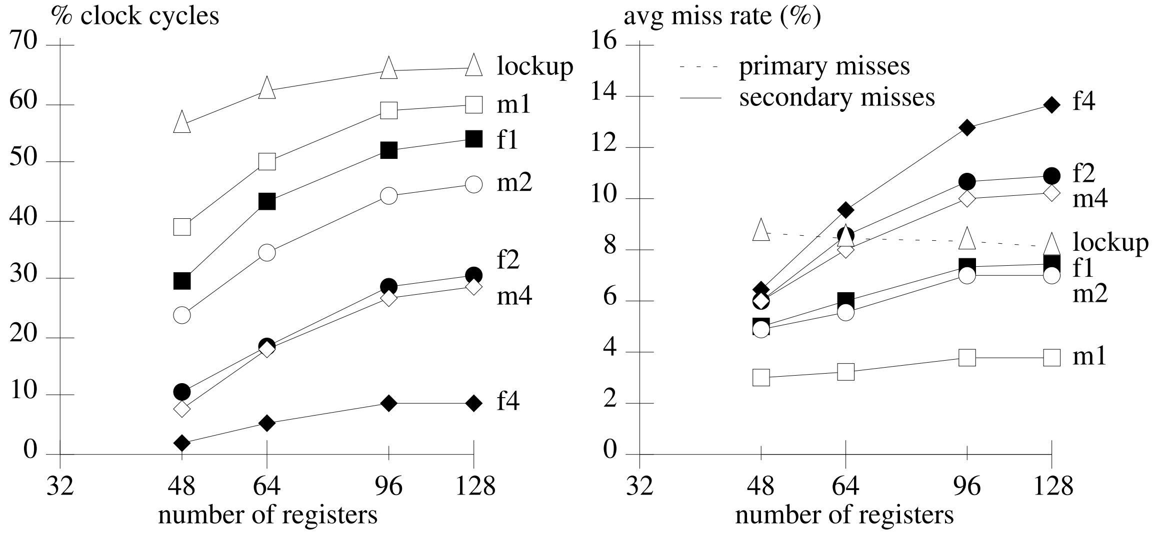
# regs	fspace	IPC values for each cache configuration								
		p	i	f4	m4	f2	m2	f1	m1	lk
256	0									
	8									
128	0	2.92	2.26	2.16	1.90	1.81	1.55	1.27	1.21	1.11
	8		2.15	2.14	1.88	1.81	1.54	1.27	1.21	
96	0	2.88	2.13	2.04	1.83	1.74	1.52	1.24	1.20	1.11
	8		2.02	2.01	1.80	1.74	1.51	1.24	1.20	
64	0	2.56	1.65	1.61	1.53	1.45	1.33	1.11	1.09	1.03
	8		1.58	1.58	1.50	1.45	1.32	1.11	1.09	
48	0	2.05	1.23	1.21	1.19	1.14	1.09	0.94	0.93	0.90
	8		1.19	1.19	1.17	1.13	1.08	0.94	0.93	

(a) 4−way issue processors

IPC values for each cache configuration									fspace	# regs
p	i	f4	m4	f2	m2	f1	m1	lk		
5.20	3.75	3.23	2.50	2.35	1.90	1.51	1.43	1.35	0	256
	3.22	3.18	2.44	2.35	1.88	1.51	1.43		8	
4.87	3.18	2.83	2.31	2.15	1.82	1.44	1.40	1.32	0	128
	2.77	2.77	2.24	2.15	1.80	1.44	1.40		8	
4.20	2.59	2.41	2.08	1.95	1.69	1.35	1.32	1.24	0	96
	2.36	2.35	2.02	1.95	1.68	1.35	1.32		8	
3.10	1.79	1.73	1.62	1.53	1.39	1.15	1.14	1.08	0	64
	1.69	1.69	1.58	1.53	1.38	1.15	1.14		8	
									0	48
									8	

(b) 8−way issue processors

Figure 5.2: Average commit IPC of all benchmarks for their (simulated) execution on the investigated systems. The commit IPC values are represented graphically by the height of the bars in the 3-dimensional bar charts, and are given explicitly in the table below each chart.



(a) Percentage of (simulated) clock cycles in which a load-miss induced stall was in effect.

(b) Primary and secondary load-miss rates. The primary miss rates for all lockup-free cache organizations are coincident with the curve for the lockup cache.

Figure 5.3: Statistics for load-induced cache misses for the four-way issue processors using an eight-cycle fetch spacing. Each figure contains a curve for each of the seven in-flight miss restrictions that were investigated.

The data presented in Figure 5.2(a) also suggests that system performance is more heavily affected by increasing the number of registers than providing support beyond four in-flight misses. For example, consider the system $(c = m_4, r = 64 : s = 8)$, that is, one with support for four in-flight misses, 64 registers, and an eight-cycle fetch spacing. Compared to this baseline system, one having twice the number of registers $(c = m_4, r = 128 : s = 8)$ will perform 25% (=$\frac{1.88}{1.50} - 1$) better, while one with no restrictions on in-flight misses $(c = i, r = 64 : s = 8)$ will perform only 5% (=$\frac{1.58}{1.50} - 1$) better.

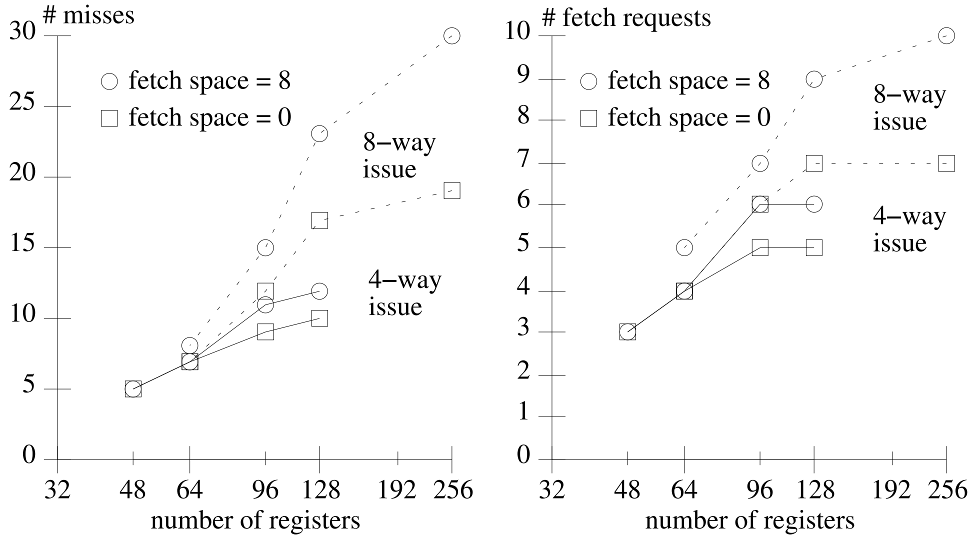
The above noted relationships between performance, number of registers, and support for in-flight misses are in part due to the increase in the number of structural-hazard-induced stalls that occur when the amount of in-flight miss support is reduced. To illustrate this increase in stalls, Figure 5.3(a) presents the percentage of (simulated) clock cycles in which such stalls occur in the systems using the four-way issue processors with an eight-cycle fetch spacing. In the figure, the percentages for each in-flight miss restriction are given by the seven curves as a function of the number of registers. (These percentages represent the average of the percentages for each benchmark.) For instance, with 64 registers, the percentage for each in-flight miss restriction given in Figure 5.3(a) is: 5% for f_4, 18% for m_4, 19% for

f_2, 34% for m_2, 43% for f_1, 49% for m_1, and 62% for a lockup cache.

The significant difference in the above percentages between f_x and m_x restrictions for the same value of x is due to the benchmarks requiring significant amounts of secondary-miss support. This requirement is suggested by the miss rates presented in Figure 5.3(b). As can be observed, while the primary miss rates for all systems having a given number of registers are sufficiently similar that the data points are coincident, there are significant differences in the secondary-miss rates. With less support for in-flight misses, the load instructions that correspond to these secondary misses must instead be held in the dispatch queue. Moreover, because memory instructions cannot be issued during a structural-hazard-induced stall, the dispatch queue will tend to fill up with memory instructions for which the data is already in the data cache. Thus, the rate at which entries are freed and new instructions are inserted will decrease as the stall progresses. Together, these two effects reduce the demand for registers and the performance gains that accompany increasing the number of registers. As the data in Figure 5.3 suggests, this reduction is more significant with larger numbers of registers because the secondary miss rates are larger, and because structural-hazard-induced stalls occur for a greater percentage of the (simulated) run-time.

The trend of decreasing performance with less lockup-free support is more pronounced with the systems having eight-way issue processors. Data supporting this trend is presented in Figure 5.2. Observe that there is less variance in the commit IPC values when more restrictions are imposed on in-flight misses in the systems with four-way issue processors (Figure 5.2(a)) than in the systems with eight-way issue processors (Figure 5.2(b)). This trend occurs because the performance of systems with wider-issue processors is more sensitive to the design of the memory system. The cause of this increased sensitivity is the ability to issue more instructions per cycle, which tends to reduce the number of clock cycles between the issuing of load instructions. As a result, at any point in the execution of an application, there tends to be a greater number of in-flight fetches and secondary misses.

Additional insight into this difference between the two issue widths and the importance of the memory system is provided by examining the average number of in-flight misses and fetches, information that is presented in Figures 5.4(a) and 5.4(b) respectively. These figures use the 90^{th} percentile in lieu of a geometric or arithmetic average owing to the non-uniform and non-Gaussian shape of the distributions



(a) 90th percentile number of misses

(b) 90th percentile number of fetches

Figure 5.4: In-flight fetch and miss averages for the systems using a lockup-free cache with no restrictions on in-flight misses.

for each benchmark[4]. Observe that for 96 and 128 registers, there are more in-flight misses and fetches when the eight-way issue processor is used. A larger number of in-flight fetches will induce more memory contention and hence a larger effective cache-miss penalty as long as the bandwidth of the interface is finite. The performance impact of this contention is reflected in the more significant change in the commit IPC values when a 0-cycle fetch spacing is used in place of an 8-cycle fetch spacing with the eight-way issue processors. This change can be observed in Figure 5.2 by comparing the height of the bars with light-gray tops, which correspond to an 8-cycle fetch spacing, and those with dark-gray tops, which correspond to a 0-cycle fetch spacing. Comparison indicates that the difference is generally greater for the eight-way issue processors. The performance of systems with less support for in-flight misses and fewer registers is less affected by fetch spacing. This trend is due to the accompanying decrease in the number of in-flight fetches that can be supported, and hence, a drop in the amount of memory contention.

Further insight into the register requirements of the different cache organizations is provided by examining the register usage histograms obtained when simulations were performed with these organi-

[4]The number of in-flight fetches (misses) is derived from normalized histograms to prevent the distribution of a single benchmark from dominating the combined distribution. The histograms used for fetches (misses) are obtained by first having the simulator record the number of in-flight fetches (misses) in each cycle of a benchmark's execution. Then, the distribution of cycle counts for each number of fetches (misses) is normalized by the (simulated) run time of the benchmark, giving normalized cycle counts of between zero and one. Next, the normalized distributions for all benchmarks for a given system design are averaged together. Finally, using this distribution, the number of in-flight fetches (misses) to cover 90% of the distribution is determined.

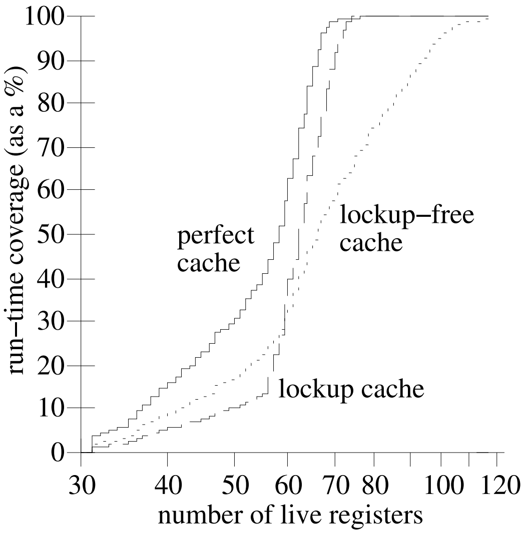


Figure 5.5: Cumulative register usage histogram for *compress* showing how many registers are live as a percent of run time. The system modeled used precise exceptions, a 4-way issue processor, 32 entry dispatch queue, and 2048 registers. The maximum value on the horizontal axis is 120 because it was very rare for more than 120 registers to be live.

zations and a register file of 2048 registers; Section 4.2 discusses the reasons for choosing such a large number of registers. The integer register histograms for *compress* were chosen as they clearly show the differences between the organizations. In addition, for clarity, the histograms for only the following three organizations are considered: the perfect cache, the lockup-free cache with no in-flight miss restrictions, and the lockup cache. These histograms are presented in Figure 5.5 as run-time coverage curves. As in Figure 4.2 (see page 68), the x-axis specifies the number of registers live at each cycle during the execution of the benchmark while the the y-axis indicates the percentage of the total number of cycles with at most the indicated number of registers live.

Comparison indicates that the lockup-free cache requires more registers to obtain the same run-time coverage as the perfect cache. In addition, the smaller slope of the lockup-free-cache curve indicates that the live registers are spread over a wider range. This range becomes smaller when a lockup cache is used, suggesting that the requirement for additional registers for the lockup-free cache is a result of allowing multiple outstanding cache misses and cache probes to occur during the servicing of the misses. The curve for the lockup cache, however, is similar in shape to that for the perfect cache, but for the lockup cache most registers are concentrated in a narrower region (between 55 and 75). This observation suggests that there is less variance in the register requirements. This reduction in the register requirements may be due to more in-order issuing of instructions and thus less variance in the time required to meet the conditions for freeing registers.

In summary, the following observations can be drawn from the above discussion:

- Systems with support for fewer than four in-flight misses have similar performance for all register-file sizes, both issue widths, and the two investigated memory bandwidths. Increasing the hardware support beyond this point provides an increasingly larger performance differential.
- Systems with a greater amount of support for in-flight misses require a greater proportion of the support to be for secondary misses, since the secondary miss rate tends to increase as the amount of in-flight-miss support is increased. Additional registers should also be provided to offset the increase in the average register lifetime that is a result of the ability to support a larger number of in-flight misses. This increase in the average register lifetime induces more variance in the time required to meet the conditions for freeing registers.
- System performance is more heavily affected by increasing the number of registers than providing support beyond four in-flight misses.
- System performance is relatively unaffected by the bandwidth of the memory interface if the processor can issue a maximum of only four instructions per cycle. But, when the issue width is increased to eight, the bandwidth of the memory interface has a more significant impact on performance, especially in systems with support for at least four in-light fetches.

5.3 Stream Buffer Implications

Stream buffers can lessen the performance impact of cache misses since they can reduce the fetch latency for missing cache blocks. To investigate the design and performance implications of stream buffers, the system model was extended to include the stream-buffer models described in Section 2.3. The revised system model was then used to evaluate the behavior of the benchmarks on a number of systems. The systems chosen (Table 5.3) capture the behavior at several representative points in the large design space.

A system is defined by selecting the parameters of the processor, the memory system, and the stream-buffer implementation from the appropriate sub-tables in Table 5.3. One system, for example, is a four-way issue processor with 48 registers, a lockup cache with a fetch spacing of 8 cycles, and a stream-buffer implementation offering unit strides and no memory-traffic filtering. As in the previous section, all systems assume a 64K byte, two-way set-associative data cache with a 32-cycle fetch latency, and all enforce precise exceptions. The 10 stream-buffer implementations (rows 2-11 in the table) each

processor parameters	
issue width	# registers
4-way issue with 32-entry dispatch queue	48 64
8-way issue with 64-entry dispatch queue	64 128

memory parameters		
cache type	miss limit	fetch spacing
lockup	lk	0 8
lockup-free	f_1 f_2 i	

stream buffer parameters				
#	abbrev.	stride predictor	traffic filters	ref. §
1	N	no stream buffers		
2	S_b	unit (baseline)		$\S_a$
3	S_i	unit	incremental fetching	$\S_b$
4	A_f	unit	P&K allocation filters	$\S_c$
5	A_{f+i}	unit	filters & incremental	$\S_b$,$\S_c$
6	A_d	min-delta	P&K allocation filters	$\S_c$
7	A_{d+i}	min-delta	filters & incremental	$\S_b$,$\S_c$
8	P	per-load		$\S_d$
9	P_f	per-load	stride filters	$\S_d$
10	P_i	per-load	incremental fetching	$\S_b$,$\S_d$
11	P_{f+i}	per-load	incremental & filters	$\S_b$,$\S_d$

Table 5.3: Systems for investigating stream-buffer implementations. A system is defined by selecting from the columns processor, memory, and stream-buffer design parameters. Note: $\S_a$ = Section 2.3.2, $\S_b$ = Section 2.3.3.1, $\S_c$ = Section 2.3.3.2, and $\S_d$ = Section 2.3.3.3.

comprise eight, four-entry stream buffers, and optional support for either memory-traffic filters, dynamic-stride prediction, or both. The implementation with none of this optional hardware is referred to as the *baseline* implementation, and for notational convenience, it is designated S_b. The abbreviations used for the other implementations are given in the table along with the sections in which they are described.

5.3.1 Implications of Unenhanced Stream Buffers

For clarity in presenting key observations, consider first the performance implications of including the baseline stream-buffer implementation in 16 of the systems listed in Table 5.3. These systems use: (1) either of the two issue widths, (2) either of the associated two register-file sizes, (3) either a lockup cache or a lockup-free cache with no in-flight miss restrictions, and (4) either a 0- or an 8-cycle fetch spacing. The performance of the benchmarks on the resulting 32 systems (16 without stream buffers and 16 with stream buffers) is presented in Figure 5.6. This figure gives the average commit IPC for each benchmark as a function of the issue width of the processor, the size of the register files, the type of data cache, the fetch spacing, and the optional use of stream buffers. The columns with N at their base (e.g., the column marked by $\boxed{1}$) give the commit IPC values for systems without stream buffers, while those with S_b at the base (e.g., the column marked with $\boxed{2}$) give the commit IPC for systems with stream buffers.

One important observation is that if stream buffers are included in a system, there is an overall performance gain, with this gain being bigger for systems having lockup caches and a larger number of

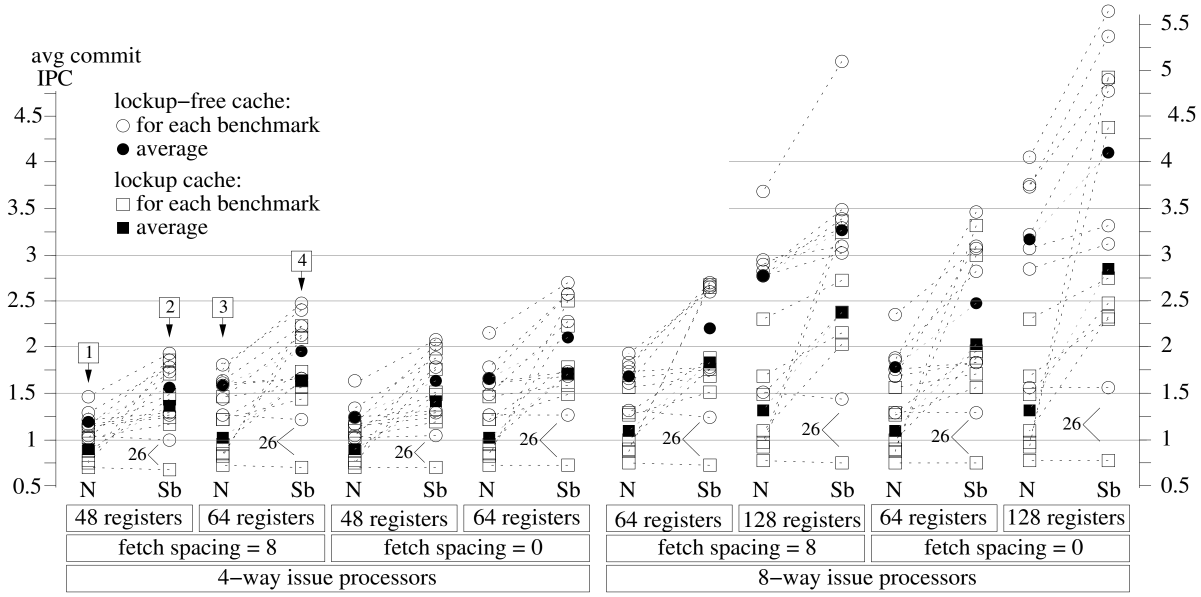


Figure 5.6: Average commit IPC for systems using either a four-way or an eight-way issue processor and either no stream buffers (N) or the baseline stream-buffer implementation (S_b). The dotted lines connect the data points of each benchmark for each cache implementation. The unfilled circles indicate the IPC for each benchmark obtained with the use of a lockup-free cache having no in-flight miss restrictions, while the filled circles indicate the overall average IPC for all benchmarks. The squares indicate similar information but for the use of the lockup cache.

registers. To illustrate this relationship, consider the columns marked with $\boxed{1}$ and $\boxed{2}$ in Figure 5.6 that correspond to the performance obtained on systems with 48 registers, a fetch spacing of eight cycles, and a four-way issue processor. Comparison of the relative position of the data points in these two columns shows that the data points are generally higher in the second column. The average increase for systems with the lockup-free cache (the filled circles) is 0.35 (=1.54-1.19), while for systems with the lockup cache (the filled squares), the average increase is 0.46 (=1.36-0.90). Expressed as speedup ratios, these increases correspond to a speedup of 1.29 (=$\frac{1.54}{1.19}$) for the systems with a lockup-free cache and a speedup of 1.51 (=$\frac{1.36}{0.90}$) for systems with a lockup cache.

The more significant speedup for systems with a lockup cache is a consequence of the hardware being blocked from issuing memory instructions during stalls induced by structural hazards. For systems with this type of cache, stream buffers enhance the performance because they significantly reduce the effective cache-miss penalty. Systems with lockup-free caches, however, can issue unrelated memory instructions during such stalls, and thus, are both less sensitive to the effective cache-miss penalty, and more tolerant of cache misses. The amount of tolerance such systems have is a function of the number of physical registers, because the number of registers determines how many instructions can be

bench-mark	speed-up	primary misses		SB
		%	% resolved by SB	% blocks used
(1)	(2)	(3)	(4)	(5)
appsp	1.09	6.0	28.9	18.1
compress	0.95	12.9	0.2	0.1
hydro2d	1.70	12.6	77.3	97.7
mdljdp2	1.06	2.4	60.6	48.5
su2cor	1.32	8.7	53.1	63.8
swm256	1.36	6.7	44.7	29.3
tomcatv	1.64	11.2	71.6	98.4

(a) dynamically-scheduled processor

bench-mark	speed-up	primary misses		SB
		%	% resolved by SB	% blocks used
(1)	(2)	(3)	(4)	(5)
appsp	not simulated			
compress	1.06	8.9	1.1	0.3
hydro2d	1.61	13.1	90.2	71.2
mdljdp2	1.03	2.7	57.4	43.0
su2cor	1.29	10.8	59.5	55.8
swm256	1.32	9.2	52.4	22.0
tomcatv	1.77	9.0	87.6	86.1

(b) statically-scheduled processor

Table 5.4: Effectiveness of the baseline stream-buffer implementation when used in (a) a dynamically-scheduled 4-way issue processor with 48 registers, and (b) the statically-scheduled 4-way issue processor described by Farkas et al. [1995].

simultaneously in execution. Thus, with more registers, the speedup obtained with stream buffers will decrease. As illustrated in Figure 5.6, when the number of physical registers is increased from 48 to 64 (columns $\boxed{3}$ and $\boxed{4}$), the speedup for systems with lockup-free caches drops to 1.25 (=$\frac{1.97}{1.58}$), while the speedup for systems with lockup caches rises to 1.58 (=$\frac{1.63}{1.03}$).

The above observations suggest that if a system can support a large number of in-flight misses, it is more beneficial to increase the number of registers than it is to include stream buffers. To illustrate this phenomenon, consider again the same four system configurations. When a lockup-free cache is used, a 32% (=$\frac{1.58}{1.19} - 1$) performance improvement is obtained by increasing the number of registers from 48 to 64, but an improvement of only 29% (=$\frac{1.54}{1.19} - 1$) is obtained by using stream buffers. On the other hand, when a lockup cache is used, a 14% (=$\frac{1.03}{0.90} - 1$) performance improvement is obtained by increasing the number of registers from 48 to 64, but an improvement of 51% (=$\frac{1.36}{0.90} - 1$) is obtained by using stream buffers.

The performance gains achieved with stream buffers are a result of the buffers prefetching a significant number of cache blocks that are used to resolve cache misses. Table 5.4(a) presents data to quantify the frequency at which cache misses were anticipated. This table provides:

- The run-time speedup due to the use of the stream buffers (column 2);
- The primary data-cache miss rate (column 3);

- The percentage of primary misses resolved using data either present in a stream buffer, or for which a fetch request had already been launched (column 4); and
- The number of cache blocks used to resolve one of these misses as a percentage of the total number of cache blocks returned by the memory system in response to a fetch request from a stream buffer (column 5).

Ignoring *compress* for the moment, observe that 18% to 98% of the blocks prefetched by the stream buffers (column 5) are used to resolve 29% to 77% of primary misses (column 4).

The statistics presented in the table are sensitive to the number of physical registers. If the number is increased, there is little change in both the number of primary misses (column 3), and the number of times data is fetched in response to a stream buffer request (column 5). But, there is a significant drop in the number of misses for which the missing data is actually present in a stream buffer. This drop is in large part due to the increase in the issue IPC that accompanies the increase in the number of registers. This increase in the issue IPC reduces the time between successive cache misses, and thus reduces the time available for prefetching. *Compress*, however, does not benefit from stream buffers. Rather, as shown in Figure 5.6 by the curves labeled "26", for a fetch spacing of 8 cycles and a four-way issue processor with 48 registers, its performance decreases by 4% ($=1 - \frac{0.66}{0.69}$) when run on the systems having the lockup cache, and by 5% ($=1 - \frac{.99}{1.04}$) when run on the systems having the lockup-free cache. This performance decrease is a result of the stream buffers prefetching mostly un-needed cache blocks, thereby delaying the launching of fetch requests that are needed to service cache misses. On these systems, less than 0.2% of primary cache misses are resolved using prefetched cache blocks. The percentages are much higher for the other benchmarks, as shown in Table 5.4(a) by the data given in column 4.

To provide further insight into the effectiveness of the baseline stream-buffer implementation, similar information to that presented in Table 5.4(a) is presented in Table 5.4(b) for the system that uses the statically-scheduled processor described by Farkas et al. [1995]. Both of these systems use the same lockup-free data cache and write-buffer organizations, the same fetch spacing and fetch latency values, the same number of functional units with the same latencies, and the same instruction scheduling rules. For the statically-scheduled processor, these rules were used during compilation, whereas, for the dynamically-scheduled processor, they are used by the dynamic scheduler.

It is important to note that compilation for the statically-scheduled processor was done using a modified version of the Multiflow VLIW Compiler [Lowney 1993], whereas for the dynamically-scheduled processor, the native Alpha 21064 compilers were used. Thus, during (simulated) execution

cache type	fetch spacing	4-way # registers		8-way # registers	
		48	64	64	128
lockup	8	1.51	1.58	1.70	1.87
	0	1.57	1.67	1.80	2.15
lockup-free	8	1.29	1.24	1.30	1.18
	0	1.33	1.28	1.38	1.29

Table 5.5: Average speedup in (simulated) run-time for all benchmarks due to the use of the baseline stream-buffer configurations.

of the benchmarks, the issue order is different for the processors owing to one processor's use of dynamic scheduling. In addition, the set of instructions issued in each case is different. With these differences in mind, observe that the speedup ratios differ by less than 10%, suggesting that similar performance gains are obtained with the inclusion of stream buffers. However, if the number of physical registers in the register files of the dynamically-scheduled processor were increased, the speedup ratios for the use of stream buffers would decrease, possibly leading to a much bigger difference in the speedup ratios for the statically- and dynamically-scheduled processors.

A second important relationship that is suggested by the data of Figure 5.6 is the performance insensitivity of the systems to the bandwidth of the interface. When the fetch spacing is reduced from 8 cycles to 0 cycles, with the exception of *compress*, the commit IPC values change relatively little. This observation suggests that even with a fetch spacing of 8 cycles, contention for the memory interface is not significant. One result of this lack of contention is that with an 8-cycle fetch spacing, when a stream buffer is re-allocated, 95 times out of 100, all of its entries contain valid data. Consequently, when the fetch spacing is reduced to zero cycles, there will not be a large increase in the amount of prefetched data, and hence, only small performance gains are likely. When the fetch spacing is changed to 0 cycles, the least significant change in the speedup ratios is an increase from 1.29 to 1.33 with 48 registers and a lockup-free cache, and the most significant change is an increase from 1.58 to 1.67 with 64 registers and a lockup cache. These speedup ratios are given in Table 5.5 under the major column heading "4-way". Like the other benchmarks, *compress* performs better with a fetch spacing of 0 cycles, but unlike the other benchmarks, it does not achieve a speedup greater than one. This phenomenon is shown in Figure 5.6 by the curves near the bottom whose right-most data points are labeled with "26". Observe that when a fetch spacing of 0 cycles is used, the curves are nearly horizontal.

When the baseline stream buffers are used in systems with an eight-way issue processor and 64 dispatch queue entries, the same trends as those noted for the four-way issue processors occur, but the differences tend to be greater. The increased significance is a consequence of the eight-way issue processor issuing and committing more instructions per cycle thereby resulting in a compression of time. This temporal compression not only reduces the time between the occurrence of cache misses but also increases the number of secondary cache misses, and the instruction cost of servicing all cache misses. Consideration of the data points presented in Figure 5.6 and the speedup ratios presented in Table 5.5 for the four-way and eight-way issue processors leads to the following observations:

- With the exception of *compress*, there is a net increase in the commit IPC for the benchmarks on systems that include the baseline stream buffers.
- This increase is more significant for systems having either more restrictive lockup-free implementations, smaller fetch spacings, or wider-issue capability.
- Increasing the number of registers results in a more significant increase in the commit IPC with systems having a lockup cache but a less significant increase with systems having a lockup-free cache. The increase is larger for the systems with the eight-way issue processors.

5.3.2 Memory Traffic Reduction and Dynamic Strides

While the baseline stream-buffer implementation improves the performance of six of the benchmarks, its effectiveness is limited by certain features of its design. One such feature is the allocation of a stream buffer whenever a data reference misses in both the data cache and the stream buffers. This policy can result in prefetching down a subsequently unused stream should the data reference be isolated, thereby generating excess traffic to the memory system. This excess traffic may degrade performance by possibly delaying the launching of fetch requests for the cache, or prefetch requests belonging to other, more useful streams. This policy of always allocating may also generate excess traffic because too frequent reallocation may cause soon-to-be-used data to be flushed from a stream buffer before it can be used. Another possible source of excess traffic is the use of only a unit stride for determining the next block to prefetch, since this stride may capture poorly, if at all, non-unit-stride miss-address patterns.

The memory traffic generated by stream buffers and the possibility of flushing useful data can be reduced using the techniques described in Section 2.3.3. The baseline stream-buffer implementation was modified to incorporate these techniques, and simulations were performed. This section presents the results and implications of the these simulations.

The design space that was investigated comprised 480 systems, which are derived from the parameters listed in Table 5.3. Each system was formed using (1) one of the four processor designs, (2) one of the four cache designs, (3) one of the two fetch spacings, and (4) one of 15 stream-buffer implementations. Of these 15 implementations:

- One corresponds to the use of no stream buffers (row 1 of the stream buffer parameters in Table 5.3),
- One corresponds to the baseline design (row 2).
- One corresponds to the baseline design with incremental fetching (row 3).
- Four correspond to the use of the PK (Palacharla and Kessler) allocation filters, the min-delta stride predictor, or both (rows 4-7) with a 32-entry miss-address table,
- Four correspond to the use of the four per-load-based designs (rows 8-11) with a 10 entry missed-load buffer, and
- Four correspond to the use of these same four designs but with 32 entries rather than 10.

The results obtained from the simulated execution of the benchmarks on these systems show a number of common relationships, and these will be illustrated using the *su2cor* benchmark. These 60 systems had an eight-way issue processor, 64 physical registers and a fetch spacing of 8 cycles. Figure 5.7(a) presents the average commit IPC values for these executions, while Figure 5.7(b) presents the total number of cache blocks that were supplied by the memory interface. In both figures, each of the curves for the four data-cache configurations (i, f_2, f_1, and lk) includes a data point for the 15 stream-buffer implementations. The stream-buffer implementations are listed along the horizontal axes using the abbreviations given in Table 5.3. However, the abbreviations used for the designs based on the per-load scheme (e.g., P_f) also include a suffix (e.g., 10) to indicate the number of entries in the missed-load buffer.

An important relationship suggested by the data presented in Figure 5.7(a) is that the enhancements to the baseline stream-buffer implementation have at best a small effect on performance, with this effect being more significant for the systems with less aggressive cache configurations. For example, when the implementations are used with the lockup cache (curve lk), there are pronounced variations in commit IPC values, but when the implementations are used with the lockup-free cache having no in-flight miss restrictions (curve i), the commit IPC values vary very little. Comparison of the data points on curve i indicates that all implementations provide similar performance with the worst performance

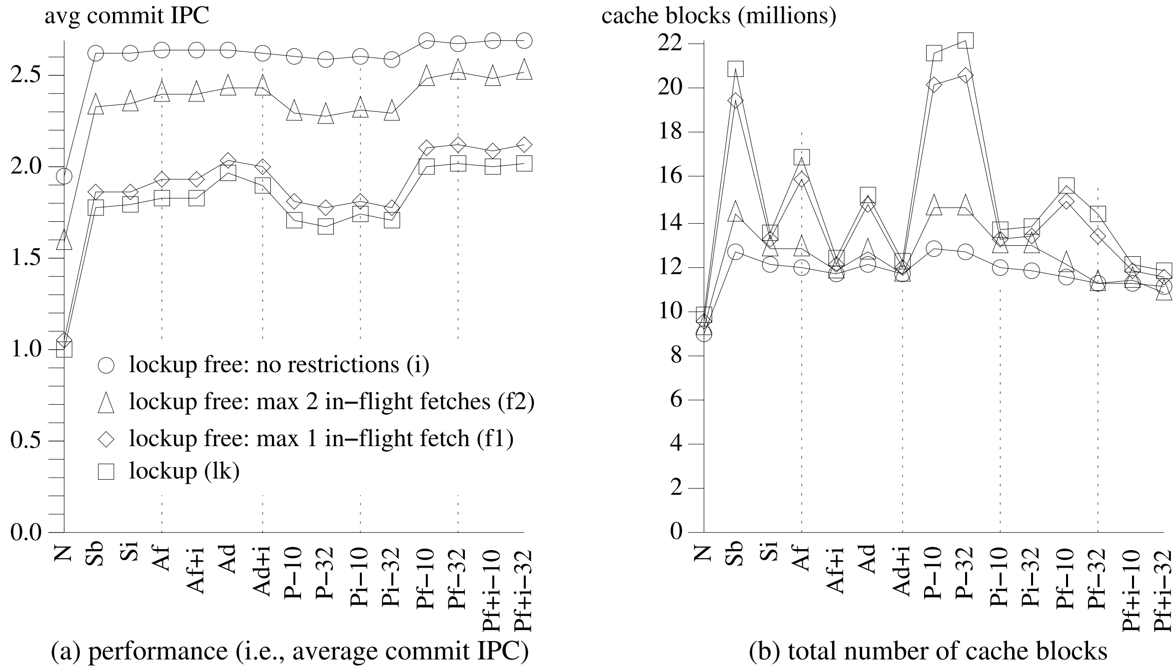


Figure 5.7: Performance of *su2cor* and the total number of cache blocks supplied by the memory interface when *su2cor* was run on systems using one of the 15 stream-buffer implementations. All systems had an 8-way issue processor, 64 physical registers, and an 8-cycle fetch spacing.

provided by the P_{-32} implementation and the best performance provided by the P_{f-10} implementation. Since the performance difference between these two implementations is 4% (=$\frac{2.66}{2.55} - 1$), clearly the stream-buffer implementation does not significantly affect the performance of *su2cor* when no in-flight miss restrictions exist. However, when restrictions on in-flight misses are imposed, the choice of stream-buffer implementation is more important, as indicated by the more pronounced variations in the commit IPC values for the other cache configurations. Comparison of the data points for these cache configurations indicates that the P_{f+i-32} implementation provides the best performance (although other implementations give similar performance) while the P_{-32} implementation again provides the worst performance. Using the data presented in the graph, the performance difference between these two implementations is 11% (=$\frac{2.49}{2.25} - 1$) with the f_2 (two in-flight fetches) cache configuration, 19% (=$\frac{2.09}{1.76} - 1$) with the f_1 cache configuration, and 20% (=$\frac{2.00}{1.66} - 1$) with a lockup cache.

Analysis of the simulation results suggests that the increased performance sensitivity of the more restrictive cache configurations to the stream-buffer implementation is due to two factors. First, systems with more support for in-flight misses have greater tolerance for cache misses. Thus, whether a stream-buffer implementation improves performance by correctly prefetching data that is missing from the cache, or degrades performance by tying up the memory interface with misfetched data, the overall performance impact, be it positive or negative, will be smaller. Second, systems with more support for in-flight misses tend to require fewer clock cycles to execute an application. Hence, there are fewer clock cycles available for prefetching data, and there is a reduction in the number of clock cycles between the detection of cache misses for which the missing data has not been prefetched. As a result of these two effects, systems with less restrictive cache configurations tend to fetch fewer cache blocks from the memory interface. This trend is suggested by the relative positions of the curves in Figure 5.7(b). The performance differential between the stream-buffer implementations is a reflection of the ability of an implementation to correctly prefetch data and to hold on to the data until it is required. Thus, the implementations with some form of filtering give better performance (Figure 5.7(a)) and generate less memory traffic (Figure 5.7(b)).

To illustrate the factors affecting the performance sensitivity of the stream-buffer implementations, consider two systems each of which uses the A_d stream-buffer implementation, one with a lockup cache, and the other with a lockup-free cache having no restrictions on the number of in-flight misses. Because *su2cor* runs 34% (=$\frac{2.60}{1.93} - 1$) faster on the system with a lockup-free cache, there are 25% fewer request-launch slots in the pipeline of the memory interface, that is, 6.8 million fewer slots; the number of slots is equal to the number of clock cycles divided by the fetch spacing. Yet, at the same time, when the lockup cache is replaced with the lockup-free cache, the number of cache-initiated fetch requests increases from 2.5 million to 3.4 million. This increase results in an increase from 9% to 17% of the number of request-launch slots now occupied by cache-initiated requests. In addition, the percentage of cache misses resolved using data that is stored in a stream-buffer entry drops from 70% to 48%, while the percentage resolved using data for which the prefetch is still in progress rises from 8% to 15%. Thus, with a lockup-free cache, there are fewer opportunities for launching prefetches, and of those launched, if the data will be used, the fetch request for it will be launched closer to the point at which the cache miss will be detected. As a result, the performance impact, be it positive or negative, of the stream-buffer implementations is less significant with the use of a lockup-free cache.

A second important relationship involves the number of physical registers and the amount of support for in-flight misses. For systems using the lockup-free cache with no restrictions on in-flight misses, the performance impact of the stream-buffer implementation is even smaller if the number of physical registers is increased. Increasing the number of registers allows more instructions to be simultaneously in some stage of execution, which increases the tolerance for cache misses, and decreases the number of clock cycles required to run the application. Thus, when the number of registers is increased, the commit IPC varies less for the various stream buffer implementations. This effect can be seen by comparing curves 1 and 2 in Figures 5.8(a) and 5.8(b). Figure 5.8(a) is a reproduction of Figure 5.7(a) for the systems with 64 registers. Figure 5.8(b) presents the IPC values for the systems with 128 registers. Observe that doubling the number of registers results in an increase of $\approx$34% (=$\frac{3.5}{2.6} - 1$) in the commit IPC.

The curves for the other three cache configurations, which have less support for in-flight misses, are also shown in Figures 5.8(a) and 5.8(b). These curves do not become flatter when *su2cor* is run on systems with more registers, but rather, show slightly more pronounced variations in the commit IPC values, especially as the support for in-flight misses decreases. The amount of support for in-flight misses, as discussed in Section 5.2 (page 88), tends to dictate how often a stall in the issuing of memory instructions will occur because there are no available miss-handling resources. When such stalls occur, there tends to be a reduction in the rate of inserting new instructions into the system, and thus, a drop in the flow rate of instructions through the system. Because the performance impact of this drop is a function of the amount of time over which a stall is in effect, stream-buffer implementations that are better at lowering the effective miss-penalty perform better.

A third important relationship is that removal of the restriction on the bandwidth of the memory interface leads to a more pronounced difference in the relative performance obtained with the stream-buffer implementations. This relationship is suggested by the data presented in Figures 5.8(c) and 5.8(d). These two figures present the average commit IPC values obtained by *su2cor* when the fetch spacing is lowered from 8 to 0 cycles. When memory bandwidth is not limited, the stream-buffer implementations that are more beneficial are those that are better at prefetching and holding onto data that is subsequently required to resolve a cache miss. Thus, while the techniques for filtering stream-buffer allocations have a positive performance impact, the incremental prefetching technique, which serves only to reduce memory bandwidth requirements, has a negative impact. However, incremental prefetching reduces the bandwidth consumed by stream buffers by 50%.

In summary, the following are the most significant results of the simulations of *su2cor* and the other six benchmarks:

- Increasing the amount of (hardware) support for in-flight data-cache misses reduces the performance differential between the stream-buffer implementations.

- Increasing the number of physical registers is likely to lead to a larger performance differential between two stream-buffer implementations if there are more restrictions on in-flight misses.

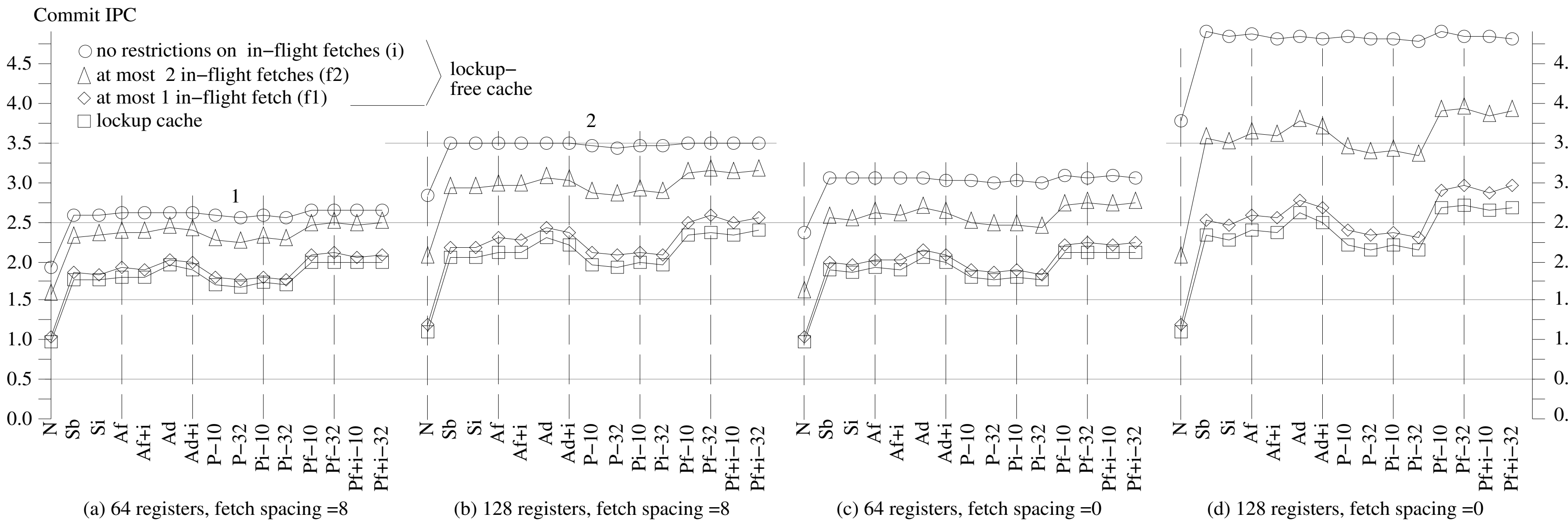


Figure 5.8: Average commit IPC values for *su2cor* when run on systems with the 8-way issue processors.

- Increasing the bandwidth of the interface increases the performance differential between stream-buffer implementations. This difference is more significant for systems with a larger number of physical registers.

- Reducing the issue width of the processor (by half) yields similar trends but the performance variations are less significant owing to the ability to issue at most half as many instructions in each cycle compared to the wider-issue processor.

- On systems with a type i lockup-free cache, the stream-buffer enhancements had less than a 5% impact on performance with the exception of *swm256*. Furthermore, these enhancements had little impact on the performance of *mdljdp2* and *hydro2d* irrespective of the cache organization, while they significantly degraded the performance of *tomcatv* if in-flight miss restrictions existed. *Appsp*, *su2cor*, and *swm256* performed best if the system included one of the following two sets of mutually exclusive enhancements: (1) the PK-allocation filters, and optionally, the min-delta stride predictor; or (2) the per-load stride predictor with filtering, and optionally, incremental prefetching. Finally, for all the systems considered and all benchmarks, the stream-buffer implementations with PK-allocation filters or stride filters generated between 1.5 and 4.2 times less memory traffic. In most cases, the per-load stride predictor with stride filtering generated the least amount of memory traffic.

5.4 Chapter Summary

This chapter has presented an investigation and analysis of the design of the register file and the other levels of the data memory hierarchy. The analysis has focused on identifying performance trends and design relationships. The components considered seek to reduce the apparent time-cost of servicing cache misses or increase the tolerance for data-cache misses. The following conclusions can be drawn from the analysis.

Similar performance was obtained from all systems having support for fewer than four in-flight misses, irrespective of the register-file size, the issue width, and the memory bandwidth. Increasing the hardware support for in-flight misses beyond this point provides an increasingly larger performance improvement for the configurations considered. However, the differentials that are obtained are smaller than those obtained by increasing the number of registers. For example, consider the system with a four-way issue processor, support for four in-flight misses, and 64 registers. Compared to this baseline system, one having twice the number of registers will perform 25% better, while one with no restrictions on in-flight misses will perform only 5% better.

Systems with a greater amount of support for in-flight misses require a greater proportion of the support to be for secondary misses, since the secondary miss rate tends to increase as the amount of in-flight-miss support is increased. Additional registers should also be provided to offset the increase in the average register lifetime that is a result of the ability to support a larger number of in-flight misses.

System performance is relatively unaffected by the bandwidth of the memory interface if the processor can issue a maximum of only four instructions per cycle. But, when the issue width is increased to eight, the bandwidth of the memory interface has a more significant impact on performance, especially in systems with support for at least four in-flight fetches. With 128 registers and a lockup-free cache having no in-flight miss restrictions, the benchmarks performed, on average, up to 15% better with an eight-way issue processor when the fetch spacing was reduced from 8 to 0 cycles, but only 5% better with a four-way issue processor. While the performance sensitivity to the bandwidth is small for all but the most aggressive systems, the reported percentages represent a lower bound due to the assumption that neither the instruction cache nor the write buffer use the memory interface. If these two components were to compete for the bandwidth of the interface, contention would increase, which would increase the performance sensitivity to the bandwidth. The rate of this increase would be greater for systems with the eight-way issue processor since wider issue processors generate more traffic in a given time period than narrower issue processors. Nonetheless, if stream buffers are not included in the system, the bandwidth is unlikely to be a significant factor for the less aggressive systems and the benchmarks discussed in this chapter.

The addition of stream buffers to a system leads to a more significant performance increase for systems having either more restrictive lockup-free caches, more memory bandwidth, or more instruction issue capability. For the systems investigated, the design of the lockup-free cache had the greatest impact on performance, and the bandwidth of the memory interface had the least impact. Increasing the number of registers results in a more significant performance increase with systems having a lockup cache but a less significant increase with systems having a lockup-free cache. The increase is larger for the systems with the eight-way issue processors.

System performance is not significantly affected when stream buffers are used that have traffic filters and dynamic-stride calculators. This observation is not surprising since the performance of the benchmarks is at best 16% better when the memory interface has an infinite bandwidth. In other words, contention for the interface is not a significant problem. This observation also suggests that the address stream generated by cache misses when each benchmark is run is dominated by interleaved unit-stride streams. Thus, the cost of supporting dynamic-stride calculation is not warranted for these seven benchmarks. However, in spite of the similar performance of many of the enhanced stream-buffer implementations, the cost of those that have traffic filters is probably warranted since the system model does not take into account all possible sources of memory traffic. The exact type of filter would have to be determined when a specific system is being considered.

This discussion concludes the investigation of hardware-based techniques for reducing the performance impact of cache misses. The investigation of the memory system is continued in the Chapters 6 and 7 with an investigation of the hardware and software complexities introduced by partitioning the register file so as to increase the performance of a dynamically-scheduled processor.

Chapter 6

Design and Implementation of the Multicluster Architecture

This chapter presents an investigation of a novel decentralized architecture for dynamically scheduled processors called the *multicluster* architecture [Jouppi]. Motivating the investigation of the multicluster architecture is the observation drawn from the work in Section 4.4 that the complexity and the size of hardware components may limit the potential performance of a processor. Therefore, hardware solutions are desired that decrease the clock cycle time while not degrading performance due to other factors. The multicluster architecture concept was initially proposed by Jouppi to address this issues, and is derived from the *centralized* or *single-cluster* architecture investigated in Chapters 4 and 5.

Although a patent has been filed on the multicluster architecture concept by Jouppi, this chapter presents the first detailed description of the architecture. This description was developed and modeled as part of this dissertation. An analysis has also been performed as part of this dissertation of the multicluster concept, and the results and implications of the analysis are presented in this chapter. Following from the analysis, static instruction scheduling algorithms have been developed, and these are described in Chapter 7.

The description of the multicluster architecture is presented in the next section. It is assumed that the reader is familiar with the operation of the single-cluster architecture. Then, Section 6.2 describes the similarities between the multicluster architecture and existing architectures. Without loss of generality, the ensuing discussions of the multicluster architecture are presented in terms of a multicluster processor with two clusters.

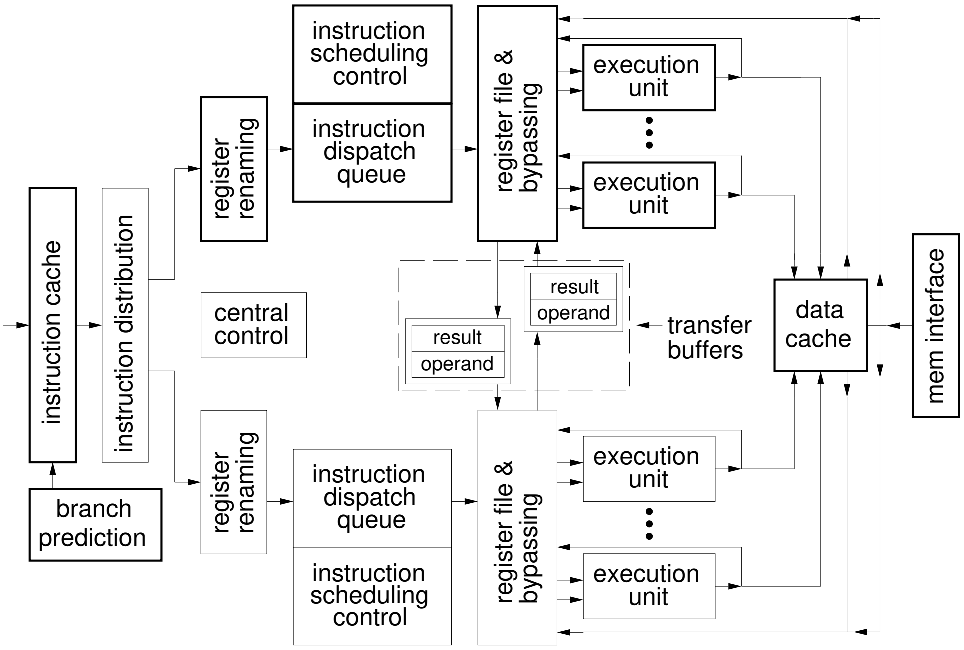


Figure 6.1: A dual-cluster processor built within the multicluster architecture framework. The components common to the single cluster architecture shown in Figure 3.1 (page 49) are outlined in bold. The write buffer and optional stream buffers are not shown, nor are many of the control and data paths.

6.1 Architecture Description

The goal of the multicluster architecture is to reduce the clock-cycle time relative to a single-cluster architecture with the same number of hardware resources, by reducing the size and complexity of the dispatch queue, scheduling logic, and register file. For example, a two-cluster processor with four execution units per cluster is contrasted with a single-cluster processor with eight execution units. Under the assumption that the register file is on a critical timing path of the processor, the cycle time of the processor can be reduced by reducing the cycle time of the register file. As discussed in Section 4.4, the cycle time of the register file is determined largely by the number of read and write ports. To reduce the number of these ports in the multicluster architecture, the number of instructions that each cluster can issue in one cycle is reduced, and access to the register file from the other cluster is disallowed.

A multicluster processor with two clusters comprises two tightly-coupled single-cluster processors that share the data and the instruction caches, and execute instructions from the same instruction stream. This commonality between the single- and dual-cluster architectures is illustrated in Figure 6.1 by outlining in bold the components that are required to implement a single-cluster processor (see Figure 3.1 on page 49). As shown in Figure 6.1, each cluster has a separate register file, a dispatch queue, and a set of execution units, and these cannot be accessed from outside the cluster. Thus, the instructions executed by the execution units of a cluster C are dispatched from the dispatch queue of cluster C, and are the only instructions that can read or write the physical registers of cluster C.

Instructions are read from the instruction cache in fetch order and are distributed, also in fetch order, to one or both clusters. If an instruction can't be distributed to a cluster because a required hardware resource is not available, the instruction stream is stalled until the resource becomes available. The distribution of instructions to the clusters is based on the architectural-register-to-cluster assignment and the registers named by each instruction. Each architectural register is assumed to be statically assigned to one or both clusters; a scheme developed as part of this dissertation for dynamically reassigning the architectural registers is described in Section 7.2.5. An architectural register that is assigned to one cluster is said to be a *local register* while one assigned to both is said to be a *global register*. The architecture imposes no restriction on the architectural registers that may be designated as global, but, as discussed in Section 6.1.2, only a small number is feasible. The zero register[1], however, does not require hardware resources (e.g., register file ports) and is thus by default designated global.

When an instruction I_1 is executed that names a local register R as a destination, the value computed by I_1 is stored in a physical register of the cluster to which register R has been assigned. When an instruction I_2 is executed that names a global register G as a destination, the value computed by I_2 is stored in a physical register of each cluster. Thus, two physical registers are required to maintain the value of a global register, one in each cluster, but only one physical register is required to maintain the value of a local register.

To execute instruction I_2, a copy of the instruction is distributed to both clusters. This *dual distribution* of the instruction provides the mechanism by which (1) a physical register in each cluster is allocated for the new value of G, and (2) the value computed by one copy of I_2 is written into the two allocated physical registers. Neither copy of I_2 duplicates the work of the other so as to limit the number of resources consumed by processing both of its copies. Instruction I_1 will also be dual-distributed if the local registers it names are not all assigned to the same cluster. In this case, dual distribution provides the mechanism by which either (1) a source operand is transfered to the cluster that performs the computation, or (2) the result of the computation is transfered to the cluster to which the destination register is assigned. If instruction I_1 or I_2 names a global register as a source operand, the value of the operand is read from the cluster on which the computation is being performed. In summary, when an instruction names a global register G, an inter-cluster data transfer is never required to read G, but one

[1] A zero register is a special architectural register with the following properties: when it is read, a value of zero is always returned, and when it is written, the value being written is discarded.

is always required to write G. On the other hand, if a local register R is named, an inter-cluster data transfer *may* be required to either read R or write R.

The performance obtained from a multicluster processor is affected by the number of instructions distributed to one cluster, the number distributed to two clusters, and how these instructions are arranged in the instruction fetch stream (because the order affects resource availability). Dual-distributed instructions require more hardware resources than instructions distributed to one cluster. Thus, the number of instructions simultaneously in some stage of execution depends on the number of single- and dual-distributed instructions. A larger number of dual-distributed instructions contributes to a smaller instruction throughput, and, hence to degrade performance. When an instruction is dual distributed, the two copies operate as a pair to perform the required task. Because the mechanism that supports this cooperation introduces some overhead, a dual-distributed instruction requires more clock cycles to execute than a single-distributed instruction (assuming all other conditions are the same). Thus, not only do dual-distributed instructions contribute to a reduction in throughput, they may also require more clock cycles to execute. However, these two negative effects are offset by the reduction in the cycle time of the processor clock that is provided by partitioning the register file and other hardware resources. Consequently, even though an application may require more clock cycles to execute on a multicluster processor than on a single-cluster processor, the run time may be reduced.

6.1.1 Instruction Distribution and Execution

Based on the architectural-register-to-cluster assignment and the registers that are named by an instruction, an instruction may be distributed to one or both clusters. An instruction I that names an (architectural) destination register R_a^d will be distributed to and hence executed on only one cluster if:

Condition 6.1 (Single cluster execution)

1. *The local registers named as source operands by instruction I are assigned to the cluster C; and*
2. *The destination register R_a^d is either a local register assigned to cluster C or a zero register.*

Evaluation of this condition for an instruction yields one of three outcomes: (1) single-cluster distribution to cluster C, (2) single-cluster distribution, but, because the instruction names no local registers and no destination register, the cluster for execution cannot be determined, or (3) dual-cluster distribution. If outcome 2 applies, the hardware will distribute the instruction to the cluster with the greatest number of free dispatch-queue entries. This selection mechanism has two advantages. First, it has a relatively small implementation cost, and second, by monitoring the relative fullness of the dispatch queues, instruction-stream stalls can be avoided when the dispatch queue of one cluster is full and the dispatch queue of the other is not full.

To execute an instruction to which outcome 3 applies, a *master copy* of the instruction will be distributed to one cluster and a *slave copy* to the other. Operating as a pair, these two copies will read the operands for the instruction, compute the new result, and write it into the allocated physical register(s). Neither copy duplicates the work of the other so as to limit the number of resources consumed by the dual issuing. To this end, for all but memory instructions, the master copy does the computation with the slave copy supplying an operand or writing the destination register. For load instructions, the master copy computes the load address, and launches a fetch to the memory system, while the slave copy receives the data and writes it into the target register. For stores, the master copy writes the data being stored into the write buffer while the slave copy computes the address. This assignment of copies to tasks seeks to ensure that the write address is not written into the write buffer long before the data is available since, the address calculation is typically done ahead of data manipulations. To determine the cluster on which the master copy of an instruction executes, condition 6.2 is applied. Cluster C executes the master copy of an instruction I if:

Condition 6.2 (Cluster determination)

1. *I is a load and names a non-global address register that is assigned to cluster C, or*
2. *I is a store and names a non-global data register that is assigned to cluster C, or*
3. *The majority of the local registers named by I are assigned to the cluster C, or*
4. *I names a local destination register that is assigned to cluster C, or*
5. *There is no majority, and cluster C has more free dispatch queue entries than all other clusters.*

To illustrate the application of this condition, consider a load instruction I that names a source register R_a^s and a destination register R_a^d. If neither R_a^s nor R_a^d are global registers and if both are assigned to the same cluster, then, instruction I will be distributed to only one cluster. But, if these two local registers are not assigned to the same cluster, the master copy of I will be distributed to the cluster to which R_a^s is assigned (point 1). However, if R_a^s is a global register, then, the master copy will be distributed to the cluster to which R_a^d is assigned (point 4). And, finally, if R_a^d is also a global register, then, the master copy will be distributed to the cluster with a larger number of free dispatch-queue entries (point 5).

Instructions in the multicluster architecture are fetched out of the instruction cache and are dispatched to one or both clusters by the instruction-distribution hardware in fetch order. Conceptually, the distribution process is envisioned to proceed as follows. After an instruction is fetched from the instruction cache, it is placed in a FIFO queue. The queue is assumed to have sufficient capacity to hold as many instructions as can be fetched out of the instruction cache in a cycle. In each cycle, both clusters determine whether the instructions in the queue should be dispatched to the cluster in question, and whether they can be. Each cluster makes this determination by inspecting the entry at the head of the queue and applying conditions 6.1 and 6.2. If the instruction is not to be executed on cluster C, then cluster C sets a flag to indicate that it has made a decision about the instruction. If, however, the instruction is to be executed on cluster C, the hardware copies it into the dispatch queue of cluster C if (a) there is a free dispatch-queue entry, and (b) a physical register is available for renaming the architectural destination register if there is one. If the instruction can be inserted, then the aforementioned flag is set. Once the flag for each cluster is set, the instruction at the head of the queue is retired from the queue, and the inspection process begins again with the next instruction. This sequential processing of instructions, however, would not be done in practice, because insertion of n instructions per (processor) clock cycle would require an insertion clock running n-times faster. Rather, sufficient hardware would be provided to reach a decision about n instructions in parallel while preserving in-order insertion.

6.1.1.1 Master and Slave Copies, and Transfer Buffers

To better explain the operation of the master and slave copies of an instruction and the use of the *transfer buffers*, consider the integer add instruction $r_2 \leftarrow r_0 + r_1$. As stipulated by condition 6.1, this instruction will be dual distributed if either of the following is true: (1) the destination register is a global register, or (2) the local registers named by the instruction are not all assigned to the same cluster. There are various scenarios under which dual distribution is required and these can be grouped into four main categories:

1. All three named registers are local registers but one of the source registers, say r_0, is assigned to a different cluster than the other two registers. Alternatively, r_1 could be a global register.

2. All three named registers are local but the destination register is assigned to a different cluster than the two source registers. Alternatively, one of the source registers could be a global register.

3. Both source registers are local registers and are assigned to the same cluster while the destination register is a global register. Alternatively, one or both of the source registers could be a global register.

4. Both source registers are local registers and are assigned to different clusters while the destination register is a global register.

For each category, the hardware must perform a different sequence of steps to execute the instruction. These steps were identified through the analysis carried out as part of this dissertation of the multicluster architecture. The steps that are performed for each category are examined below using a scenario from each category.

First, suppose that all named registers are local registers but source register r_0 is assigned to cluster C_2 while source register r_1 and destination register r_2 are assigned to cluster C_1; this scenario is included in the first category. To perform the add, the slave copy is dispatched to cluster C_2 to make the value of r_0 available to the master copy. The master copy is dispatched to cluster C_1, and when it is inserted into the dispatch queue of cluster C_1, r_2 is renamed using a physical register belonging to this cluster. When executed, the master copy computes the add and writes the result into the physical register bound to r_2. The slave copy is not allocated a physical register because the destination of the instruction is a local register that has been assigned to cluster C_1.

The slave copy has an input dependence induced by the immediately preceding instruction in fetch order that wrote r_0, and, thus, the slave copy cannot be issued until after this dependence is resolved and when an integer issue slot is available. The hardware requires an integer issue slot to issue the slave copy because the slave copy must read the value of r_0 from the integer register file, and to do so requires access to a read port. The master copy has an input dependence induced by the immediately preceding instruction that wrote r_1, and it also has a *copy dependence* induced by the slave copy. This copy dependence guarantees that the master copy will be issued only when its second operand, that is r_0, is available. Therefore, the master copy will be issued after both input dependences are resolved and when a suitable functional unit is available.

In the write-back pipeline stage (discussed below) the slave copy writes the value of r_0 into an entry in the *operand transfer buffer* of cluster C_2, the master copy's cluster. The operand transfer buffers are shown located between the clusters in Figure 6.1. An entry in the operand transfer buffer is allocated to the slave copy when it is issued. If an entry is not available, then the slave copy is blocked from being issued, and in certain circumstances, an instruction-replay exception is required to avoid issue deadlock; Section 6.1.1.2 discusses this issue in more detail. When the slave copy writes the value into the allocated entry, it also stores the unique ID of the instruction of which it is a copy. This ID is used by the hardware to associatively search the operand transfer buffer to locate the operand for the associated master copy. The copy dependence between the master copy and the slave copy is removed when the slave copy is issued, thereby permitting the master copy to be issued as soon as the next cycle. If the master copy is issued in the next cycle, it obtains the value of r_0 from the forwarding network rather than

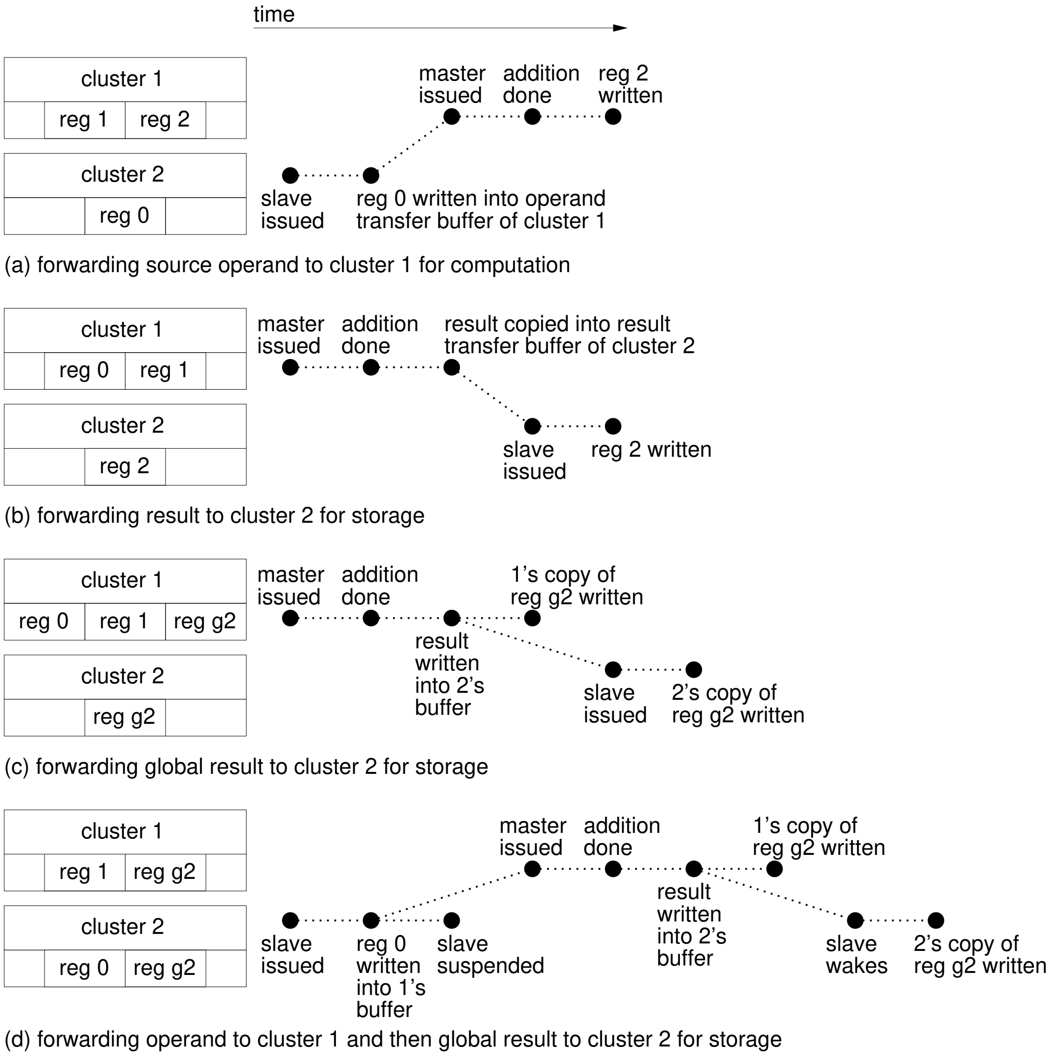


Figure 6.2: Dual execution of the instruction $r_2 \leftarrow r_0 + r_1$ with the architectural registers assigned to the indicated clusters.

the operand transfer buffer; an example of operand bypassing is presented below. After the master copy obtains the value from either the forwarding network or the buffer, the entry allocated to the slave copy is freed. This entry can be used by another instruction in the next cycle. The steps required to execute the add instruction for this scenario are shown in Figure 6.2(a).

The second scenario is similar to the first with the exception that the two source registers are assigned to cluster C_1 while the destination register r_2 is assigned to cluster C_2; this scenario is included in category 2 of the category list on page 111. As before, all three registers are uniquely assigned to a cluster, that is, they are local. Because the two source operands for the operation are located on the same cluster, the master copy of the instruction will be issued first. The result is computed, and then forwarded to the slave copy. The slave copy is then issued and it writes the forwarded result into the physical register bound to r_2. Unlike in the first scenario, the physical register is allocated to the slave copy because the destination register is assigned to the slave copy's cluster C_2 and not the master copy's cluster C_1. The data transfer is performed by writing the value computed by the master copy into an entry in the *result transfer buffer* of cluster C_2. This entry is freed after the slave copy reads the result out of the entry prior to writing the result into the bound physical register. The steps for this scenario are shown in Figure 6.2(b). To prevent the slave copy from being issued before the result is available, a copy dependence is created between it and the master copy. This dependence is removed two cycles before the master copy is due to finish computing the result, and, thus, for simple one-cycle latency instructions like the add, the slave copy can be issued as soon as one cycle after the master copy is issued. As before, if the slave copy is issued in the next cycle, the data is provided to it by the forwarding network and not the allocated buffer entry.

The third scenario is similar to the second scenario with the exception that the destination register r_2 is a global register and not a local register; this scenario is included in category 3 of the category list on page 111. Because the destination is a global register, both the slave and master copies are allocated physical registers when they are inserted into their respective dispatch queues. The same sequence of steps as for the second scenario category is performed to execute the add, with the exception that, when the master copy writes the result into the slave copy's result transfer buffer, the master copy *also writes the result into the physical register it was allocated*. Figure 6.2(c) illustrates this scenario.

The fourth scenario differs from the third in that both source registers are not assigned to the master copy's cluster C_1. Rather, one of the source operands, say r_1, is assigned to cluster C_1; this scenario is included in category 4 of the category list on page 111. As in the first scenario, the slave copy is issued only after its data dependence is resolved and when both an integer issue slot and an operand transfer buffer entry are available. Once the slave copy writes the operand into the allocated entry, the hardware suspends it. The master copy is then issued only after its data dependence is resolved, and when both a suitable functional unit and a result transfer buffer entry are available. When the master copy obtains the forwarded source operand, it frees the operand transfer buffer entry (which is associated with its cluster). It then computes the result, and writes it into both the allocated entry (which is associated with the slave copy's cluster) and into its own register file. Finally, the slave copy is awakened, it obtains the result, frees the result transfer buffer entry, and writes the result into its own register file. Figure 6.2(d) illustrates the steps for this scenario.

Execution Pipeline and Bypassing Details

As with the single-cluster processor, the multicluster processor is assumed to have a four-stage execution pipeline: in the first stage an instruction is issued, in the second, its source operands are provided, in the third, it is executed, and in the fourth, its result is written to a physical register, to a transfer-buffer entry, or in the case of a store, to the write buffer. With the exception of the execution stage, all stages are assumed to have a single-cycle latency. Thus, an instruction I that is executed by a single cluster in a functional unit with latency L will require $L + 3$ cycles to execute. If this instruction generates a result, the result is available for bypassing by the end of the $(L + 2)^{th}$ cycle.

The cycle Y in which an instruction I can be issued is equal to $\max(Y_{op} - 2, Y_{dual})$, where Y_{op} and Y_{dual} are clock-cycle counts. Y_{op} is the cycle at which the operands used by the instruction are available; the -2 indicates that registers are read at the end of the second stage, that is, in cycle $Y + 1$. Y_{dual}, on the other hand, is applicable only to dual-distributed instructions because it specifies the cycle in which the second-to-issue copy of an instruction can issue. If execution of instruction I requires that the slave copy forward an operand to the master copy, the slave copy will be issued first, and the master copy can be issued in following cycle, that is, $Y_{dual,master} = Y_{slave} + 1$. However, if the master copy forwards a result to the slave copy, the slave copy can begin *execution* in the cycle after the master copy writes the designated transfer-buffer entry. Thus, if the master copy writes the entry in cycle Y, then $Y_{dual,slave} = Y - 2$.

To illustrate these timing relationships, consider the three-instruction sequence shown in Figure 6.3(a). Under the assumption that registers r_0, r_2, and r_4 are assigned to cluster C_1, and register r_3 is assigned to cluster C_2, the first instruction will be executed only by cluster C_1. The second instruction will be dual executed, with its master copy being executed by cluster C_1 before its slave copy is executed by cluster C_2. The third instruction will also be dual executed, but its slave copy will be executed by cluster C_2 before its master copy is executed by cluster C_1. If it is assumed that the multiply functional unit has a latency of four cycles, and the add unit has a latency of one cycle, these instructions require at least 14 cycles to execute; their cycle-by-cycle execution is illustrated in Figure 6.3(b).

6.1.1.2 Avoiding Issue Deadlock

As noted in the discussion of the first scenario, the unavailability of an operand transfer buffer entry may lead to the issue process becoming deadlocked. To avoid deadlock, the hardware detects when deadlock might occur and forces an instruction-replay exception. The conditions under which deadlock may occur are:

#	instruction
1	r2 ← r0 x r2 ; executes on cluster C1
2	r3 ← r2 x r4 ; master executes (on C1) before slave
3	r2 ← r3 + r0 ; slave executes (on C2) before master

(a) Three−instruction sequence

cycle	execution−pipeline stage				
	issue	read op	execute	write	
0	inst #1				
1		inst #1			
2					
3			inst #1		
4	master #2				
5		master #2			value of r2 bypassed to master #2
6				inst #1	r2 valid
7			master #2		
8	slave #2				
9	slave #3	slave #2			result bypassed to slave #2 for storage in r3
10	master #3	slave #3	slave #2	master #2	slave #2 reads value, writes r3, and forwards it
11		master #3	slave #3	slave #2	r3 valid & slave #3 copies value into transfer buffer
12			master #3	slave #3	value of r3 bypassed to master #3
13				master #3	r2 valid

(b) Cycle−by−cycle execution of the three instructions

Figure 6.3: A three-instruction sequence and its cycle-by-cycle execution under the assumption that instruction 1 is executed by only one cluster, and instructions 2 and 3 are executed by both clusters.

Condition 6.3 (Issue deadlock)

1. *Instruction I cannot be issued because there are no free operand buffer entries, and*
2. *All the entries have been allocated to instructions that occur after I in fetch order.*

Deadlock may arise under these conditions, because an operand transfer buffer entry can introduce a dependence between an instruction I and one earlier in the fetch order, hence inducing a circular dependence that can only be broken with the freeing of an entry. However, if point 2 of condition 6.3 is false, some of the in-flight instructions that hold operand buffer entries will eventually complete, thereby freeing the operand buffer entries, and allowing the circular dependence to be broken. The following example illustrates how deadlock can occur when point 2 is true.

Consider the three instruction sequences shown in Figure 6.4(a) in which five local registers are named: r_8 and r_9 are assigned to cluster C_1, and r_0, r_1, and r_2 are assigned to cluster C_2. Based on the registers named by each instruction and the assumed architectural-register-to-cluster assignment,

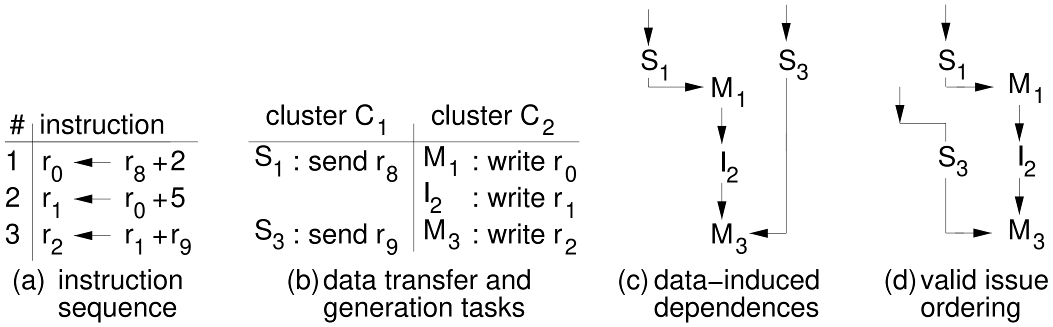


Figure 6.4: Instruction sequence which could lead to issue deadlock if registers r_8 and r_9 are assigned to cluster C_1 and registers r_0, r_1,and r_2 to cluster C_2. S_x is the slave copy of instruction x and M_x is its master copy.

instructions 1 and 3 are issued on both clusters, and instruction 2 is issued only on cluster C_2, as illustrated in Figure 6.4(b). The data and copy dependences give rise to the following relationships:

- The master copy of instruction 1 (M_1) can be issued only after the slave copy of instruction 1 (S_1) is issued,
- Instruction 2 (I_2) can be issued only after M_1 has been completed,
- The master copy of instruction 3 (M_3) can be issued only after I_2 has been completed, and
- The slave copy of instruction 3 (S_3) can be issued as soon as register r_9 is valid. Note that S_3 has no knowledge of the register written by I_2 nor the dependence induced by I_2, because the register that is written is assigned to the cluster on which S_3 does not execute, namely cluster C_2.

These relationships are shown in Figure 6.4(c).

Now, with no loss of generality, assume there is only one operand transfer buffer entry, and consider the issue ordering shown in Figure 6.4(d). When S_1 (the slave copy of instruction 1) is issued, it is allocated the operand transfer buffer entry belonging to cluster C_2; this entry can be freed only after M_1 (the master copy of instruction 1) is issued and has been provided with the operand. Because S_3 also requires the same operand transfer buffer entry to pass the value of register r_9 to its master copy, S_3 is blocked from being issued until M_1 is completed. Now, instead of the illustrated ordering, suppose S_3 is issued before S_1, as might happen if the load instruction that writes register r_8 incurred a cache miss. In this case, S_1 cannot be issued until M_3 is issued because the issuing of M_3 is required to free the operand buffer entry. But, this master copy cannot be issued before I_2 has been completed, I_2 cannot be issued before M_1 is completed, and M_1 cannot be issued until S_1 is issued. Hence, there is a circular dependence. This circular dependence *might* occur when the slave copy of an instruction I requires an operand transfer buffer entry, and all the entries are held by slave instructions that occur later in fetch order. Whether this condition actually leads to deadlock cannot be determined without considering the relationship between all instructions. As this task is formidable for even moderate numbers of instructions and a few operand transfer buffer entries, the hardware must assume that deadlock is inevitable. Thus, an instruction-replay exception is initiated to remove all instructions following instruction I from the system. In the above example, an exception would therefore be raised in the cycle in which S_1 is blocked from being issued due to there being no free operand transfer buffer entry, thereby causing the removal of I_2, S_3, and M_3, and thus freeing the operand buffer entry.

Note that issue deadlock does not arise from result transfer buffer entries because the slave copy of an instruction cannot have a dependence induced on it by an instruction later in fetch order. Thus, the slave copy is guaranteed to be eventually issued, thereby guaranteeing that the entry is eventually freed. Separate result and operand transfer buffers are provided in part to avoid such dependences and thereby to reduce the number of instruction-replay exceptions. Separate buffers also reduce the implementation cost.

6.1.2 Tradeoffs in Implementation of the Register Files

The design of a multicluster processor, as with the design of a single-cluster processor, represents a balance between meeting cycle-time and performance targets, and the required hardware complexity. Contributing to hardware complexity is the number of functional units, the maximum number of instructions that can be issued per cycle, the number of dispatch-queue entries, the number of physical registers, and the design of the memory system, factors which have been investigated in Chapters 3 to 5. In addition to these factors, the design of a multicluster processor is also affected by the hardware support for global registers and for partitioning the register files. This section considers the implications this support has on performance and on the design of the register file.

To determine whether an instruction should be dispatched to a cluster, in evaluating Condition 6.1, the hardware must determine whether the destination register is global. While the hardware cost of detecting whether a register is global is small, other hardware and performance concerns make it too costly to implement more than a few global registers. These concerns arise from the need to dispatch the instruction to both clusters, which has the effect of increasing the pressure on the physical registers, the dispatch queue entries, and the instruction issue slots. As a result, a greater percentage of the processor's resources will be used in processing dual-issued instructions, thereby increasing the number of clock cycles required to run an application. To compensate for this negative impact on performance, a wider issue width is required, which in turn requires additional register file ports. Increasing the number of ports increases the cycle time of the register file, thus possibly increasing the processor's cycle time, and therefore reducing the performance advantage of wider issue and more global registers.

The performance of an application is affected by the number of resources used to execute dual-issued instructions, and additionally by the time required to execute these instructions. A dual-issued instruction I_d takes more clock cycles to complete because it must transfer an operand or a result between clusters. In the best case, the execution of I_d will take one cycle longer than if the instruction was single issued. More typically, additional latency is induced because the two copies are unlikely to issue at the optimal times. For example, in Figure 6.2(a), if the issuing of the master copy of the instruction is delayed due to an execution-unit conflict or the unavailability of a transfer buffer entry, the production of the new value of register 2 will also be delayed. The delayed completion of a dual-issued instruction will delay the issuing of instructions that depend on its result, thereby lengthening the time required to complete the thread of computation. Furthermore, because the hardware schedules around such delays, there is likely to be more disorder in the issuing and completion of instructions. This disorder may in turn increase the number and the impact of exceptions that are caused by either mispredicted branches or contention for the operand transfer buffers.

Increased dual execution also increases the need for more transfer-buffer entries because each dual executed instruction requires either an operand transfer buffer entry, a result transfer buffer entry, or both. Having a larger number of entries permits a greater number of dual-executed instructions to be in-flight. However, the design of these buffers can impact performance in two ways. First, the number of read and write ports dictates how many instructions can simultaneously access a result or operand transfer buffer; if the number of such instructions exceeds the number of ports, some of these instructions would have to stall. Such stalls increase the effective instruction latency for dual-issued instructions, a factor that could be compensated for by increasing the number of ports.

A second source of performance degradation is instruction-replay exceptions that are required to avoid possible deadlock in the issuing of instructions. Although the frequency and hence impact of such exceptions can be reduced by increasing the number of transfer buffer entries, doing so increases the hardware complexity because these buffers must be fully associative to allow the master or slave copies of different instructions to issue out of order.

6.2 Related Architectures

The multicluster architecture shares features with existing or proposed processors. In this section, the *multiscalar* [Sohi 1995], *Multiflow* [Lowney 1993], and *simultaneously multithreaded* [Tullsen 1996] architectures are examined for their similarities and differences to the multicluster architecture. These architectures are examined in this section of the dissertation as they represent a mixture of implementation frameworks, from dynamic scheduling, to decentralization, to static scheduling with very long instruction words. More conventional dynamically scheduled processors were examined in Chapter 2 and are discussed further in Section 7.1.

6.2.1 The Multiscalar Architecture

A key motivation for the multicluster architecture is the speed increase that can likely be obtained through decentralization. Decentralization may also be used to improve performance by allowing a processor to speculatively execute blocks of code rather than individual instructions, thereby increasing the potential instruction-level parallelism. An architecture of this type is the *multiscalar* architecture proposed by Sohi et al. [1995]. Sohi et al. envision the architecture to comprise a number of processing units with a sequencer that assigns tasks to processing units; a proposed implementation is shown in Figure 6.5. The tasks are assigned dynamically as the processor speculatively traverses the control flow graph[2] of a single application, in task-sized steps, without inspecting the instructions within a task. The control flow graph is encoded in the application's object code along with information to facilitate communication between tasks. Tasks may be a part of a basic block, one or more basic blocks, a loop iteration, a function call, etc. Once a task is assigned to a processing unit, the unit fetches and executes the instructions of the task until the task is complete.

The architecture implements a single logical register file with each processing unit having its own copy. Register updates are dynamically routed among the many processing units with the aid of compiler-generated masks encoded in the object code. These masks contain bits indicating whether a given instruction should forward its result to other processing units for use by other tasks. To maintain the appearance of sequential execution, each processing unit issues the instructions assigned to it in sequential order, and, at the task level, sequential order is imposed by organizing the processing units in a circular queue. Tasks are retired only if they were being executed by the processing unit at the head of

[2] A control flow graph represents the control flow of an application with the nodes of the graph corresponding to basic blocks, and the edges corresponding to control dependences [Aho 1986].

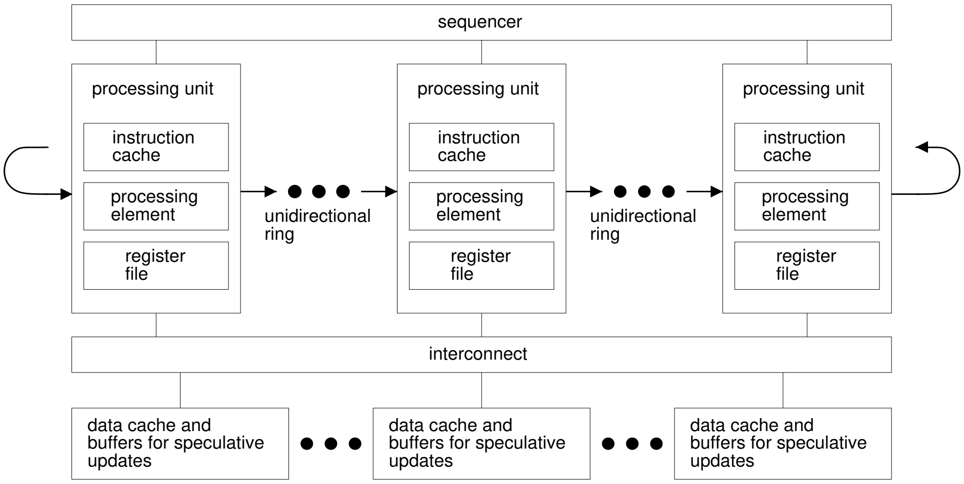


Figure 6.5: A possible implementation for a multiscalar processor.

the queue, and new tasks are dispatched only to the unit at the tail of the queue.

Two key features of the multiscalar architecture are (1) the need to predict the direction of only those branches that span tasks, and (2) the use of the control flow graph to allocate statically determined tasks to processors. The tasks being referred to here may be executed in parallel and may represent a large number of dynamic instructions.

Analysis of the above two features suggests that a much larger number of instructions might be in execution compared to the number that might be in execution with a multicluster processor or more conventional superscalar and very long instruction word processors. Yet, since each processing unit is statically scheduled, the multiscalar architecture is less tolerant than the multicluster architecture of cache misses, inter-task branches, and inter-instruction data and memory dependences. The multicluster architecture can achieve higher utilization of issue slots but at the cost of more complexity within the clusters. Another point of comparison is that both architectures provide a single register-name space, but the one provided by the multicluster architecture is simpler. Both architectures share the common need for good static scheduling of the application to keep all functional units busy. But, scheduling for the multicluster architecture is made more complex by the use of dynamic scheduling.

6.2.2 The Multiflow Architecture

The multiscalar architecture, as mentioned above, implements a single logical register file with each processing unit having a complete copy of this file. An alternative to duplication of the register file is to partition it, as is done in the multicluster architecture and the Multiflow architecture [Lowney 1993]. Among other similarities of these architectures is the partitioning of the functional units that operate

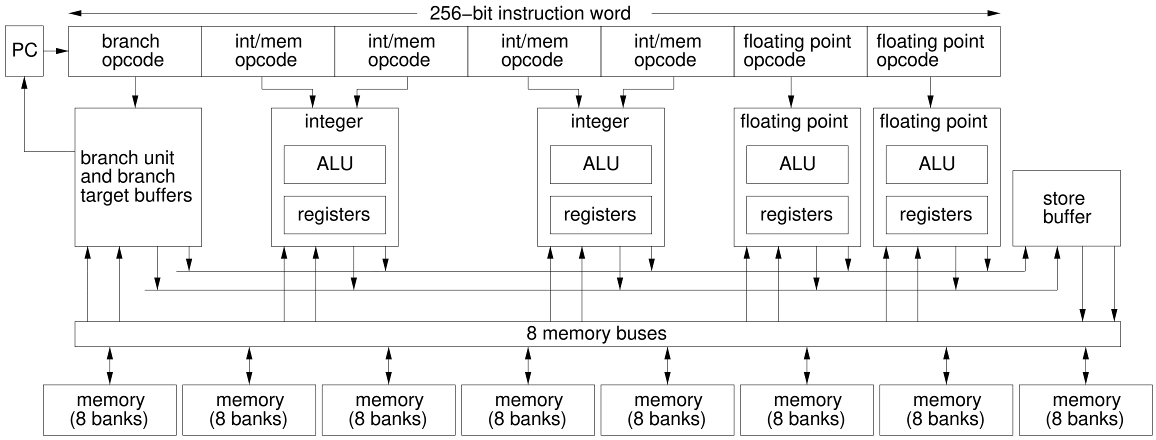


Figure 6.6: The Multiflow Trace 7/300 computer.

on instructions contained in a single instruction stream. The Multiflow architecture, however, offers no hardware interlocks or other mechanisms for resolving conflicts, with the exception of a mechanism to stall the machine in the event of a memory bank conflict. Rather, the architecture relies heavily on the compiler to statically schedule the hardware including execution units and data paths. This control is encoded in very long instruction words. For example, the Multiflow Trace 7/300, which is shown in Figure 6.6, has 256-bit instruction words that control the issuing of up to seven operations in a cycle. To uncover sufficient instruction level parallelism, the compiler uses a number of techniques including speculative execution, loop unrolling and trace scheduling.

Analysis of the Multiflow architecture revealed that, as in the case of the multicluster architecture, it distributes the registers and execution units across the clusters of the machine. On the other hand, in the multicluster architecture, a mechanism exists for accessing the registers in other clusters, in the Multiflow architecture, for an ALU (arithmetic logic unit) A to use a value stored in the registers associated with another ALU, this value must first be explicitly copied to a register in the register file of A. This two step process is coordinated by the compiler, a fact that underlines that the hardware is completely predictable, which is not true for the multicluster architecture.

One implication of the predictability of the Multiflow architecture is that it allows the Multiflow compiler to encode all the control directives for the machine into a single instruction stream while avoiding the complications that arise in the multicluster architecture from such a combined instruction stream. Further comparison of the Multiflow and multicluster architectures is presented in Section 7.2.4.4.

6.2.3 Simultaneous Multithreaded Architectures

The multicluster architecture has a single instruction-fetch stream, which is divided between multiple processing units on the basis of the registers named by each instruction. This use of a single instruction stream differentiates a multicluster processor from simultaneously multithreaded processors and tightly-coupled multiprocessors. Multithreaded processors are capable of simultaneously executing sequences of instructions, called threads, that are both control and data independent. Multithreading provides a means for increasing the utilization of the hardware since the hardware can be simultaneously used by several threads. This capability requires autonomous instruction streams for each thread. Simultaneous multithreading is reported by Tullsen et al. [1996] to be relatively inexpensive to implement on top of a superscalar (essentially) eight-way issue processor without significantly degrading the performance of a single thread. Such an architecture is reported by Tullsen et al. to yield better performance than the superscalar processor alone as long as multiple threads can be made to run simultaneously. In contrast, the aim of the multicluster architecture is to decrease the cycle time of the processor to permit a single thread of execution to run faster.

6.3 Chapter Summary

This chapter has presented the results of an analysis of the multicluster architecture concept along with the first comprehensive description of the architecture. This work was carried out as part of this dissertation.

The multicluster architecture is a decentralized, dynamically-scheduled architecture in which the register files, dispatch queue, and functional units of the architecture are distributed across multiple clusters. The motivation for partitioning these resources is to reduce the size and complexity of components that are likely to be on the critical timing path of a centralized processor, and in the process, to reduce the cycle time of the decentralized processor. Each cluster of a multicluster processor is assigned a subset of the architectural registers, and this assignment is used as the basis for distribution of the instructions of an application to the clusters. The architecture offers a mechanism to allow the register file of one cluster to be accessed by instructions being executed on another cluster. This mechanism is based on making copies of an instruction, and distributing a copy to each cluster involved in the transfer.

However, the instruction-duplication mechanism incurs a performance cost. Instructions that are duplicated require more hardware resources than those that are not. Therefore, the number of instructions that are simultaneously in some stage of execution depends on the number of those that have been duplicated. As a result, instruction throughput is indirectly related to the number of duplicated instructions. Furthermore, when an instruction is duplicated, the copies operate as a pair to perform the required task. Because the mechanism that supports this cooperation introduces some overhead, a duplicated instruction requires more clock cycles to execute than one that has not been duplicated (assuming all other conditions are the same). Thus, duplicated instructions not only contribute to a reduction in throughput, they may also require more clock cycles to execute. However, these two negative effects of duplication are offset by the reduction in the cycle time of the processor clock that is provided by partitioning the register file and other hardware resources. Consequently, even though an application may require more clock cycles to execute on a multicluster processor than on a single-cluster processor, the run time may be shorter.

To achieve a performance improvement, however, (static) instruction scheduling algorithms are required to ensure that the hardware resource requirements of the instruction-fetch stream at run time are balanced. The algorithms that have been developed for the architecture are discussed in the next chapter.

Chapter 7

Instruction Scheduling and the Multicluster Architecture

This chapter presents an analysis of static instruction scheduling for dynamically-scheduled processors. From this analysis, instruction scheduling algorithms have been developed for the multicluster architecture, and these are also described and evaluated in this chapter. The analysis, and the development and evaluation of the algorithms have been performed as part of this dissertation. The goal of this chapter is to develop a better understanding of the challenges dynamic scheduling presents to architects of compilers who must design algorithms that allow the hardware to be used to full advantage by software. The discussion presented in this chapter assumes a familiarity with the operation of the multicluster architecture, which has been described in Chapter 6.

This chapter begins with an analysis of static instruction scheduling for dynamically-scheduled processors. Section 7.2 follows with a discussion of the additional scheduling challenges posed by the multicluster architecture, and the algorithms developed for addressing these challenges. Then, Section 7.3 presents a performance evaluation of these algorithms, which raises the question of whether the properties of the multicluster architecture are such that it is a viable framework for microprocessors. Finally, Section 7.4 reconsiders the objectives of instruction scheduling for dynamically-scheduled processors in light of the understanding gained from developing the scheduling algorithms for the multicluster processors.

As in Chapter 6, without loss of generality, the ensuing discussions are presented in terms of a multicluster processor with two clusters.

7.1 Instruction Scheduling for Superscalar Processors

Prior to executing an application on a statically-scheduled or dynamically-scheduled processor, the machine-level instructions of the application are ordered; this process is known as *(static) instruction scheduling*. For a statically-scheduled processor, the goal of instruction scheduling is to minimize the number of stalls of the execution pipeline and to issue the maximum number of instructions per cycle. This goal, however, cannot be applied to a dynamically-scheduled processor. Meeting this goal requires the ability to statically predict the order in which instructions issue, and the issue order of a dynamically-scheduled processor is not deterministic. This section presents an analysis of the factors that preclude the adoption of this goal for dynamically-scheduled processors. The analysis also considers the role that static instruction scheduling plays in determining the performance of existing dynamically-scheduled processors, and the factors that can be taken into account to improve the ability of statically-generated code to fully utilize the available hardware.

7.1.1 Instruction Scheduling for Statically-Scheduled Processors

The goal of instruction scheduling for statically-scheduled processors, as noted above, is to minimize the number of pipeline stalls and to issue the maximum number of instructions per cycle. To address this goal, the instruction scheduler (i.e., the compiler) attempts to statically order the instructions so that pipeline stalls will not occur at run time due to unavailable data or hardware resources. Because the availability of data and hardware resources depends on the interaction of instructions as they are issued and executed, the compiler relies on being able to predict these run-time interactions. These predictions are often performed with the help of a model of the processor, with some models assuming constant latencies for all operations (e.g., [Lowney 1993]) while some others seek to account for variances caused by data-cache misses (e.g., [Abraham 1993, Kerns 1993]). Typically, the interaction predictions are only made in relation to instructions that belong to a given basic block because the issue order of these instructions can be determined with 100% accuracy.

To take into account the interactions between instructions belonging to different basic blocks, various techniques can be used to increase the effective size of the basic blocks, and thus, increase the number of instructions that are jointly scheduled. *Trace scheduling* [Lowney 1993] and *software pipelining* [Allan 1995, Lam 1988] are two such techniques. Trace scheduling consists of first building an *instruction trace* by unrolling loops and grouping together basic blocks that are likely to be executed in sequence, then scheduling the instructions associated with the trace as if they belonged to a single basic block, and, finally, inserting extra instructions to ensure correct behavior if the entire trace is not executed. This process is repeated until the entire application is scheduled. Software pipelining is also used to schedule loop bodies, but rather than unrolling the loop body and replicating it in the program binary, the loop body is rewritten so that it concurrently executes operations from multiple iterations of the original body. The reader is referred to [Rau 1993] for an in-depth survey of instruction scheduling algorithms for statically-scheduled processors.

7.1.2 Instruction Scheduling for Dynamically-Scheduled Processors

The goal of instruction scheduling for statically-scheduled processors cannot be applied to dynamically-scheduled processors because, unlike in the case of statically-scheduled processors, the instructions in a basic block are not guaranteed to be issued in fetch order. Rather, instructions are issued out of fetch order when in-order issue is prevented by an unresolved hardware-resource-induced or data-induced dependence. More precisely, the processor attempts to issue instructions from later in the fetch order that are not affected by the unresolved dependences. Therefore, the instruction-issue process will stall only if no unaffected instructions are found. To prevent these stalls from occurring, the compiler must be able to predict the state of the processor at each point during the execution of the application. That is, the compiler must be able to statically identify the instructions concurrently in execution, and the stage of execution of each. This information indicates the availability of hardware resources and data, and thus, indicates whether a stall might occur. But, this information cannot be statically predicted because it depends on the instruction issue order, and the issue order is not deterministic.

The non-deterministic instruction issue order is a result of several non-statically predictable factors. For dynamically scheduled architectures like the multicluster architecture or the MIPS R10000 architecture [Gwennap 1994a], these factors are:

1. The order in which instructions are considered for insertion into the dispatch queues, that is, the fetch order.

2. Physical register and dispatch-queue entry availability. These dictate whether an instruction can be inserted into a dispatch queue. If there are fewer instructions in the dispatch queue, there will be less disorder in the issuing of instructions, and thus a more deterministic ordering.

3. Memory-location dependences, because static memory disambiguation cannot detect all possible conflicts, especially in languages that support pointers.

4. Data cache misses. These delay the issuing of instructions connected to the instruction that missed by either a data dependence or a memory-location dependence.

5. Functional unit conflicts. These delay the issuing of instructions, the effect of which is an increase in the disorder of instruction issue, in the pressure on the physical registers, and possibly in the branch misprediction rate.

6. Mispredicted branches and other sources of exceptions. These mandate the removal of instructions from the processor. Resolution of an exception can dramatically change the utilization of all resources.

For dynamically-scheduled processors built within other frameworks, a similar set of factors exist. For example, items 1, and 3 to 6 above apply to the PA-8000 [Gwennap 1994b], the PowerPC 604 [Song 1994] and the Intel P6 [Gwennap 1995] processors, which are built within the framework of reorder buffers.

Because the state of a dynamically-scheduled processor cannot be statically predicted, the compiler cannot make any decisions based on the relative order in which instructions will be processed. In other words, the inner workings of the processor must be ignored. Consequently, the goal of instruction scheduling is to ensure an uninterrupted supply of instructions to the processor. More specifically, the goal is to minimize the number of stalls of the instruction-fetch stream so as to maximize the size of the pool of instructions waiting to be issued. In addition, as is the case for statically-scheduled processors, a related goal is to minimize the number of mispredicted branches so as to maximize the number of correctly-speculated instructions in the instruction-fetch stream. Maximizing the pool size increases the likelihood that the dynamic scheduler will be able to issue the maximum number of instructions each cycle. By minimizing the number of mispredicted branches, a larger percentage of the issued instructions will be committed.

The misprediction of branches and the resulting impact on performance can be reduced by judiciously ordering the basic blocks within a program binary, by changing the sense of conditional operations, and by moving unconditional branches out of the frequently executed paths [Calder 1994]. These transformations, and in particular the last one, may also reduce the number of stalls of the instruction fetch stream that are induced by control hazards. Stalls of this type occur in the PowerPC 604 and other processors. They arise when the branch prediction hardware cannot determine the target address of a branch, as might occur when predicting the outcome of a branch that does not use program-counter relative addressing. Stalls of the instruction-fetch stream are also induced by structural hazards, that is, conflicts arising from the sharing of hardware resources. One source of such conflicts is the storage that some dynamically-scheduled processors use to rapidly recover from mispredicted branches. For example, the MIPS R10000 has storage for only four branches and thus if a fifth is encountered in the instruction stream while four earlier branches are still unresolved, the instruction stream must be stalled.

A second possible source of resource conflicts is the storage used for rename registers. In processors like the PowerPC 604 and the PowerPC 620, structural hazards arise whenever there is no rename register available for an instruction. Reducing the pressure on the rename registers entails restructuring the code to reduce the time required to meet the exception model requirements. However, the MIPS R10000 and the PA-8000 do not have this limitation because there are other hardware components that impose lower bound on the number of in-flight instructions.

A third source of resource conflicts is the dispatch-buffer entries. As an example, consider the PowerPC 604 processor, which is shown in Figure 2.1 (page 17). The PowerPC 604 has a dispatch unit that dispatches instructions in fetch order to one of five reservation stations, each of which has two entries. Now suppose that the reservation station for the floating-point unit is full and a floating-point divide instruction D is the next instruction to be dispatched. In this situation, the dispatching of instructions must be stalled until one of the floating-point reservation stations is freed. During this time, other instructions that occur later in the instruction stream, that is, behind the divide D, will also be prevented from being dispatched, even though there may be reservation station entries capable of receiving them. Thus, during the instruction stream stall, the average number of instructions issued per cycle is likely to drop. To minimize the occurrence of such stalls, the compiler should distribute the instructions targeting a particular reservation station evenly in the instruction-fetch stream rather than grouping them together. That is, the instruction stream should be balanced. Like the PowerPC 604, the Intel P6 processor is also sensitive to the mix of instructions in the instruction stream, but unlike the PowerPC 604, it is the instruction decoders and not the reservation stations that may induce stalls. The Intel P6 has three instruction decoders, which are located between the instruction cache and the reservation station. Each decoder converts a unique class of x86 instructions into one or more micro-operations to enable the processor to execute the x86 instruction.

This need for a balanced mix of instructions in the instruction-fetch stream is shared by the MIPS R10000 and the PA-8000. The MIPS R10000, shown in Figure 2.4 (page 23), has three dispatch queues, one for integer instructions, one for floating point instructions, and one for load/store instructions. Because instructions are inserted into the queues in fetch order, an instruction stream stall will occur if the next instruction to be inserted cannot be inserted because the target queue is full. The R10000 has greater tolerance for instruction stream stalls than the PowerPC 604, because the R10000 has many more entries in its dispatch queues than the PowerPC 604 has entries in its reservation stations. Thus, the R10000 has a potentially much larger pool of instructions that are waiting to be issued and a stall must persist for much longer to deplete this pool. The PA-8000, like the MIPS R10000, also has greater tolerance for instruction stream stalls, because the two halves of its partitioned reorder buffer each have 26 entries; one half of the buffer holds load/store instructions, while the other holds all other instructions.

Conclusion

In conclusion, from the above-described analysis of static instruction scheduling, the following hypothesis is put forward: the non-determinism induced by the ability to issue instructions out of order in a dynamically-scheduled processor requires that the compiler ignore the internal operation of the processor, and concentrate only on ensuring the supply of instructions to the processor is uninterrupted at run-time. Further, this instruction-scheduling goal can be met by balancing the functional-unit and resource requirements of the instructions in the instruction-fetch stream. The premise on which the instruction-scheduling hypothesis is based is as follows: if the size of the waiting-to-issue pool of instructions is sufficiently large and diverse, then the dynamic scheduler will be able to find enough instructions to issue in each cycle so as to keep instructions flowing through the system at the maximum rate. In contrast, when code is scheduled for a statically-scheduled processor, the compiler has direct control over the order in which instructions issue. Static instruction scheduling is considered further in Section 7.4.

7.2 Instruction Scheduling for the Multicluster Architecture

As in the case of the processors discussed in the previous section, when generating code for a multicluster processor, the compiler must ignore the internal operation of the processor and focus only on ensuring the supply of instructions to the processor is uninterrupted at run-time. Static instruction scheduling plays an important role in ensuring that the code takes full advantage of the hardware. At the same time, the partitioning of the architectural registers introduces a source of instruction-fetch-stream stalls that is difficult to address statically. This source of stalls is not present in other superscalar processors. In this section, a discussion is presented of the complexities this new source of stalls poses for static scheduling. The section also describes novel algorithms that have been developed as part of this dissertation to address these complexities.

7.2.1 Scheduling Fundamentals

There are two requirements for an application to perform well when executing on a multicluster processor. First, the instructions that are required to perform the task must be balanced across the clusters, with each cluster concurrently performing similar amounts of work. However, this requirement cannot be directly addressed by the compiler because the work done by a cluster is a function of the order in which instructions are issued, and the issue order is not deterministic. Thus, the compiler can only indirectly address the requirement of workload balance by ensuring that the distribution of instructions to clusters is balanced. The second requirement is that the number of instructions distributed to more than one cluster must be minimized. Instructions distributed to only one cluster are preferred because dual distribution increases hardware-resource requirements and execution latency.

It is hypothesized that to meet the above two requirements for good performance, the compiler must micro-manage the allocation of architectural registers to the data values used by each instruction. This focus for compilation is proposed because the distribution of instructions is determined by the architectural registers named by each instruction and the cluster(s) to which each architectural register is assigned. The objective of choosing the architectural registers that will be named by the instructions is to generate an ideal code schedule with the following properties.

Property 7.1 (Ideal code schedule) *Let N_d be the number of instructions of an application that are distributed to both clusters for execution when the application is run, N_s^1 be the number distributed to only cluster C_1, and N_s^2 be the number distributed only to cluster C_2. A well performing code schedule will result in the following at run-time:*

1. *At no time during execution should the absolute difference between N_s^1 and N_s^2 be larger than β, the instruction-balance factor, where $\beta > 0$.*
2. *N_d should be as small as possible.*

The performance of an application whose code schedule satisfies property 7.1 is strongly affected by the value chosen for the instruction-balance factor β. Large values of β favor a single cluster over an equal distribution of instructions. As a result, the application is likely to be concurrently using only a subset of the available hardware resources. Hence, compared to a single-cluster processor with the same overall resources, more clock cycles will be required to execute the application. On the other hand, small values of β will likely lead to a larger number of dual-distributed instructions than for larger values of β. Because dual-distributed instructions require more hardware resources and have a large latency, the number of clock cycles is again likely to be larger. Thus, optimal performance demands optimal selection of the instruction-balance factor.

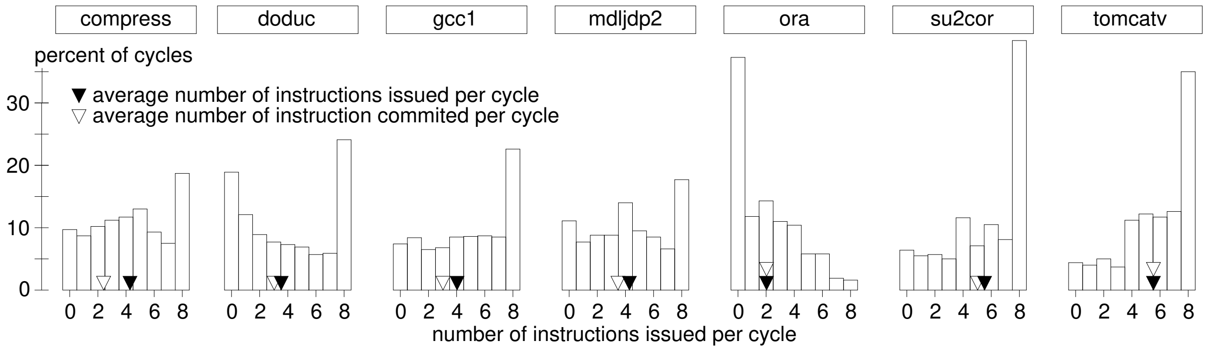


Figure 7.1: Instruction-issue distributions as a percentage of the total number of clock cycles for a single-cluster eight-way issue processor with 128 physical registers, 128 dispatch-queue entries, and a lockup-free cache with a 16-cycle cache-miss latency and a 0-cycle fetch spacing.

Experimental evidence suggests that smaller values of β and a more even distribution is preferable. Figure 7.1 shows the per-cycle instruction-issue distributions as a percentage of the total number of (simulated) clock cycles for several benchmarks. The benchmarks were run on a single-cluster processor with 128 physical registers and 128 dispatch-queue entries (the processor is described in more detail in Section 7.3.1). In spite of the significant instruction-issue disorder provided by the large number of registers and dispatch-queue entries, the dynamic-scheduler cannot find sufficient numbers of instructions to fully utilize all the issue bandwidth, as indicated by the average number of issued and committed instructions per cycle both being much less than eight. Since the benchmarks do not fully utilize the issue bandwidth of a single-cluster processor, if they were run on a dual-cluster processor with the same aggregate resources, it is unlikely that issue bandwidth would be a critical resource as long as both clusters are used. Both clusters must be used because each cluster has half the issue bandwidth of the single-cluster processor (to permit a reduction in the number of register file read/write ports), and the benchmarks can take advantage of more bandwidth than is offered by a single cluster. Moreover, to achieve similar instruction per cycle (IPC) values, a similar amount of disorder in instruction issue is required, and thus, the benchmarks must use registers from each cluster.

The above observations suggest that the primary objective of static instruction scheduling is to generate a code schedule, which, when run, generates an instruction stream in which the instruction-distribution imbalance never exceeds β. If the imbalance is bounded by β, then the distribution is said to be balanced. The secondary objective is to minimize the number of dual-distributed instructions. To address the two objectives, the compiler considers each (static) instruction individually, and seeks a register assignment for the data values used by the instructions. The criterion used to select a register and hence a cluster for a data value is whether the instruction distribution is likely to be balanced. If it

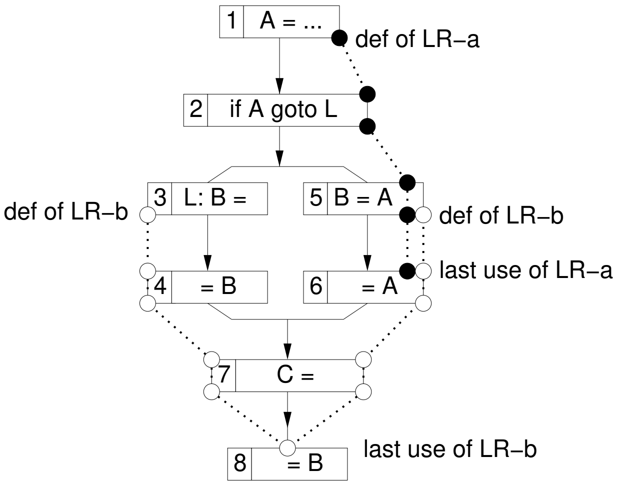


Figure 7.2: The control flow graph of a small segment of code annotated with black circles to show the live range for variable A, and with white circles to show the live range for variable B.

is not, the compiler must select a register that is assigned to the under-subscribed cluster. But if it is, then the compiler can select a register that allows the instruction to be distributed to only one cluster. To make this selection, the compiler must take into account the inter-dependences between instructions that arise from one or more instructions using a value generated by another. A useful abstraction for discussing this source of dependences is that of a *live range*.

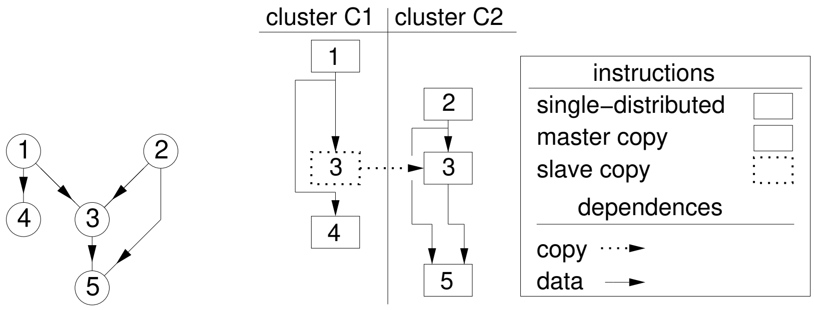
A live range is a set of points that define the instructions over which a value is live [Aho 1986]. Points are imagined to exist before and after each instruction. Thus, a value is considered live from the point immediately after the instruction that defines it to the point immediately before the last instruction to use it. The instructions that are included in a live range are those that lie on the control flow path(s) between the definition(s) of the live range and last use(s).

To illustrate these principles, consider the control flow graph (CFG) shown in Figure 7.2 for an eight-instruction segment of code. The nodes in the CFG represent instructions while edges indicate the control-flow relationship between pairs of instructions. Figure 7.2 indicates that the live range for variable A comprises the point after instruction 1, the points before and after instructions 2 and 5, and the point before instruction 6. Further, the figure indicates that the live range for variable B includes two definitions (instructions 3 and 5), in contrast to the one definition for variable A (instruction 1). Two definitions are included because the value generated by each of these definitions reaches a common use, namely instruction 8. The live ranges for variables A and B are said to *interfere* because there exists at least one point in the program at which they overlap.

With the *live range* concept in mind, a more complete examination of instruction scheduling can be

(1) $A \leftarrow 5$
(2) $B \leftarrow 23$
(3) $C \leftarrow A + B$
(4) $D \leftarrow A \times A$
(5) $E \leftarrow B + C$

(a) Instructions



(b) Data-flow graph (c) Distribution for $\beta = 1$

Figure 7.3: Code scheduling example. In (c), solid white boxes are used to represent instructions that will be distributed to only one cluster and to represent the master copies of dual-distributed instructions, while dotted boxes represent the slave copies of dual-distributed instructions. Solid directed arcs indicate data dependences, while dashed directed arcs represent the copy dependence between master and slave copies of dual-distributed instructions.

undertaken. The objective of instruction scheduling is to obtain first a balanced run-time distribution of instructions, and then a minimum number of dual-distributed instructions. The compiler tries to meet these objectives by allocating the live ranges named by the instructions of an application to the architectural registers. To illustrate this process, consider the instruction sequence shown in Figure 7.3(a) where $A, \ldots, E$ are live ranges. As depicted in Figure 7.3(b), instructions 3 and 5 depend directly and indirectly on the results of instructions 1 and 2, whereas instruction 4 depends only on instruction 1. Suppose now that the compiler is using a very large instruction-balance factor, that is, $\beta \gg 1$. To prevent any instructions from being dual distributed, all live ranges must be assigned to the architectural registers belonging to one cluster. Such an arrangement satisfies the requirements stipulated by Condition 6.1 for single-cluster execution.

On the other hand, if $\beta = 1$, the ideal distribution of instructions would be to have odd and even numbered instructions execute on clusters C_1 and C_2 respectively, with no dual-distributed instructions. However, owing to the data dependences in the given instruction sequence, some instructions will have to be dual distributed. Thus, for $\beta = 1$ the minimum number of dual-distributed instructions is one, and this minimum is obtained by allocating live ranges A and D to cluster C_1, and live ranges B, C, and E to cluster C_2. The distribution outcome of these allocations is illustrated in Figure 7.3(c). This figure, shows that instructions 1, 2, 4, and 5 are distributed to only one cluster, and instruction 3 is dual distributed.

7.2.2 Scheduling Intricacies

In the code-scheduling example of Figure 7.3, the objective was to choose a register assignment and thus a cluster assignment for the live ranges that would result in a static schedule with the ideal characteristics stipulated by property 7.1. For this example, it was assumed that the register assignment was applied to an ordered sequence of instructions rather than to a graph, and that all live ranges were allocated to non-global registers. This section discusses the rationale for these assumptions and the impact they have on the resulting schedule. The discussion seeks to provide insight into the intricacies of statically scheduling instructions for a multicluster processor.

Consider first the assumption that the starting point was an ordered sequence of instructions rather than a graph. A graph, by definition, represents only the direct dependences between instructions, such as the producer-consumer relationship between instructions 1 and 3 in Figure 7.3(b). However, to reason about instruction distribution and whether it is balanced, information is required to establish an ordering between instructions that are not dependent on each other. Thus, for example, the instruction scheduler must know whether, at run time, instruction 2 will be fetched and dispatched to a cluster before or after instruction 1 is fetched and dispatched. The scheduler must also know that instructions 1 and 3 are separated by instruction 2. This information is implicit in an ordered sequence of instructions, and it is for this reason that the allocation of live ranges to registers is carried out using an ordered sequence.

A live range might be considered for allocation to a global register if the live range is frequently used and infrequently defined. A global register might, for instance, be useful for storing the value of the stack pointer since the stack pointer is typically read more often than it is written. In using a global register G for such a live range, an inter-cluster data transfer is never required to read the live range, but one is always required to write G. On the other hand, if a local register R is used, to read or write the live range, an inter-cluster data transfer *may* be required. Whether a global or local register is more suitable for a live range depends on the relative performance impact of writing the global register G versus reading or writing the local register R. While the performance impact of global writes is *directly* related to the number of writes, the performance impact of local reads (writes) is *indirectly* related to the number of reads (writes).

The read (write) relationship is only indirect for a local register R because whether an inter-cluster data transfer is required depends on the other registers named by each instruction that names R as a source (destination). This indirect relationship severely complicates the identification of candidates for global registers. To evaluate the merits of using a global register for a live range L, most if not all of the other live ranges must already have been allocated to registers, and the choice of these registers may be dependent on the register chosen for L. Thus, this decision between using a local register or a global register for L is an example of a case where a circular dependence exists between the object about which a decision is to be made and the information on which the decision is based. In such cases, owing to the intractability of revisiting decisions when the information on which they are based subsequently changes, an algorithm must be used that makes a decision using only the information available at the time the decision is made. These algorithms are said to be greedy.

A greedy algorithm must also be used, owing to another instance of a circular dependence, when allocating live ranges to local registers. In this case, the dependence arises between the instruction distributions on which is based the selection of a local register for a live range L, and the registers subsequently chosen for some of the remaining unallocated live ranges. For convenience, let these remaining unallocated live ranges be the set $\mathcal{L}$. The dependence arises because the registers chosen for the live ranges in $\mathcal{L}$ may sufficiently change the instruction distributions on which the selection for L is based so that a register in a different cluster should have been chosen for L. That is, based on the information known after allocating the live ranges in $\mathcal{L}$, it is possible that a different register should have been originally chosen for L. However, if the compiler were to reallocate L, the instruction distributions on which the register selection of already allocated live ranges were based may change sufficiently to warrant the reallocation of some of these live ranges. While such backtracking may result in a more balanced overall instruction distribution, it is computationally expensive and possibly intractable.

Irrespective of whether backtracking is used, the allocation of a live range to a register is complicated by having to take into account the instruction distribution in the vicinity of each instruction that defines and/or uses the live range. It is possible that the allocation of the live range to a register R will increase the imbalance around some of these instructions while decreasing it around others. When this scenario arises, a greedy algorithm is required to choose a register and hence a cluster for the live range.

7.2.3 Scheduling Methodology

Through analysis of the relationships described in Sections 7.2.1 and 7.2.2, a scheduling methodology was developed. This methodology is described in this section.

To use property 7.1 as a basis for scheduling of instructions for the multicluster processor, it is required to first generate a (static) ordering of the instructions since, without an ordering, it is unclear how to measure the amount of instruction balance. To better estimate the amount of instruction balance, generation of the schedule should occur late in the compilation process. Instruction balance is maintained by categorizing the live ranges as candidates for either local or global registers, and allocating the live ranges to registers of the appropriate type, all the while seeking to minimize the number of dual-distributed instructions. However, instruction distribution cannot be directly controlled, because distribution is a function of register usage and the instruction inter-dependences arising from sharing of live ranges. In view of these facts, the following five-step approach has been developed to schedule instructions.

1. The application is compiled into an intermediate language (IL) to which are applied conventional optimizations like common expression elimination and constant propagation.

2. The IL instructions are arranged into a (static) code schedule. The IL instructions correspond one-to-one to the machine-level instructions of the processor, but unlike the machine-level instructions, the IL instructions name live ranges and not registers. The purpose of instruction scheduling is to assign live ranges to architectural registers, and thereby convert IL instructions into machine-level instructions.

3. Live ranges associated with the stack pointer and the global pointer[1] are designated as candidates for global registers; all other live ranges are designated as candidates for local registers.

4. The live ranges that are candidates for local registers are then partitioned to minimize the number of dual-distributed instructions and maximize the concurrent utilization of both clusters.

5. The live ranges are allocated to the architectural registers with global-register candidates allocated to global registers and local-register candidates allocated to local registers.

Steps 1, 2, 4, and 5 correspond to the four scheduling problems: *code optimization*, *code scheduling*, *live range partitioning*, and *register allocation*. Although solutions to these four problems must handle the unique characteristics of the multicluster architecture, the research in this dissertation focused on solving the third problem, live range partitioning, since this problem captures most of the idiosyncrasies of the architecture. The novel techniques developed as part of this dissertation for solving this third problem are described in Section 7.2.4.

The code optimization problem can be solved using techniques such as those that are well described by Aho et al. [1986] and Bacon et al. [1993]. Since this problem arises early in the compilation of an application, solutions to it are relatively independent of the unique characteristics of multicluster architecture. Thus, to limit the scope of the research, the existing techniques were used without modification.

[1]The stack pointer is used to access a procedure's local variables and arguments. The global pointer is used to access statically declared objects such as global variables and constants. The reader is referred to [Patterson 1995] for more details.

As part of this dissertation, solutions were developed to the code scheduling and register allocation problems by modifying and extending existing techniques to address the unique characteristics of the multicluster architecture. These solutions are presented in the following sub-sections.

7.2.3.1 Code Scheduling

Among the existing techniques for generating a static code schedule are basic-block scheduling [Smotherman 1991], superblock scheduling [Chang 1991], and trace scheduling [Lowney 1993]. An analysis of scheduling techniques that could be used with the multicluster architecture was carried out as part of this dissertation. From the results of the analysis, it was concluded that while various techniques could be used, techniques such as trace scheduling and superblock scheduling are preferable. These techniques have a tendency to generate large basic blocks. These techniques are preferable due to the necessity to measure instruction imbalance on a per basic-block basis when performing the live range partitioning. With small basic blocks, where 5 to 20 instructions are typical [Rau 1993], while each basic block may appear balanced or nearly balanced at compile time, at run-time, the ensuing instruction-fetch stream may be quite unbalanced. The lack of balance is due to the instruction imbalance of the stream at run time being a function of the order in which the basic blocks appear in the stream, and the static imbalance of each. To ensure that the degree of imbalance at run time never exceeds the value chosen for the balance factor β, the compiler would have to take into account all control flow paths by which each basic block can be reached. Thus, while it is not desirable to measure imbalance on a per basic-block basis, a per-basic-block solution is mandated by the complexity of examining multiple control flow paths, and by the interdependence between instruction imbalance and instruction distribution.

7.2.3.2 Register Allocation

Once a partitioning has been obtained (step 4), the live ranges are allocated to architectural registers. The goal of this process is to determine a mapping between the live ranges that have been assigned to a cluster and the architectural registers of the cluster. The mapping between live ranges and an architectural register can be many-to-one provided that live ranges do not interfere with each other. Live ranges that interfere must be allocated separate registers. If too many live ranges interfere, there may not be enough registers associated with the designated cluster. In this situation, the live range must be spilled, that is, mapped to an available register in the other cluster (if the live range is a candidate for a local register), or mapped to a memory location. Spilling, however, may alter the instruction balance since a live range L may be used by multiple instructions, each of which may use other live ranges. Thus, when L is reassigned, the instruction imbalance may change at more than one point in the schedule. To minimize the impact of live-range spills, live ranges are ranked by the number of times they are read and written, and those with the lowest ranking are spilled first.

A number of researchers including Chaitin et al. [1981], Hendren et al. [1993], and Proebsting et al. [1992] have proposed techniques to solve the register allocation problem. Of the proposed techniques, the most popular are based on the *graph coloring* abstraction. Techniques that use this abstraction build a graph in which the nodes represent live ranges and the edges represent interference. There is an edge in the interference graph from a node n_1 to a node n_2 if the live range L_1 interferes with the live range L_2, that is, both live ranges are simultaneously live and thus cannot occupy the same register. The live ranges that interfere with a live range L are called the *neighbors* of L. Using this abstraction, the register allocation problem becomes that of coloring the graph with k colors such that neighboring nodes have different colors. By choosing k to match the number of architectural registers, a k-coloring will represent a feasible allocation of live ranges to registers. An optimal allocation is not feasible, because the problem of finding a k-coloring with $k \geq 3$ is NP-complete [Briggs 1992]. Heuristic techniques are used to rank the nodes so that the higher priority ones are colored first, and to select a live range to be spilled should there be no color available for a node. Should spilling be necessary, the graph is modified, and another attempt is made at coloring it. This process repeats until a k-coloring is obtained.

Through an analysis performed as part of this dissertation of the techniques based on graph coloring, the technique developed by Briggs et al. [1994] was identified as being the most suitable for the multicluster architecture. This technique was modified to address the unique characteristics of the multicluster architecture. The technique developed by Briggs et al. is based on the approach of Chaitin et al. [1981], but it separates the process of coloring nodes from the process of spilling live ranges. The separation of these two phases provides a convenient framework for implementing the desire to spill a live range first to a local register in the other cluster and, if no register is available, then to memory. In addition, the separation of the phases increases the likelihood that a live range will be allocated a register [Briggs 1994]. This feature is desirable as it competes against the increased likelihood that more live ranges will be spilled since each cluster of the multicluster architecture is allocated only a subset of the architectural registers. More details of the register allocator that was developed for the multicluster architecture are presented in Section A.2.4.

7.2.4 Live Range Partitioning

This section describes three techniques that were developed as part of this dissertation to solve the live range partitioning problem for the multicluster architecture. These techniques are: the *min-cut scheduler*, the *global scheduler*, and the *local scheduler*. Simply stated, these techniques solve the following problem: determine a cluster assignment for each live range embodied in the intermediate-language representation of an application such that the instruction balance and the number of dual-distributed instructions meet the specified objectives.

Live range partitioning applies only to live ranges that are candidates for local registers, but in partitioning these live ranges it is sometimes necessary to take into account the existence of those that are candidates for global registers.

7.2.4.1 The Min-cut Scheduler

The min-cut scheduler solves the following problem: given an assignment of architectural registers to clusters, find an architectural-register assignment for the live ranges that minimizes the number of instructions at run-time that are dispatched to both clusters. The technique developed to solve this problem uses an approach based on graph partitioning. This approach is possible since a balanced distribution of instructions is not required. Hence, there is no need to take into account ordering-induced inter-instruction dependences. Therefore, a schedule-based solution is not mandated (see Section 7.2.2).

The min-cut scheduler begins by building a graph from the intermediate-language representation of an application. In this graph, there is a node for each live range and an edge for each instruction. An edge corresponding to an instruction I interconnects the nodes that correspond to the live ranges that instruction I reads or writes. In graph-theoretical terms, such an edge is known as a *hyper-edge* [Gibbons 1985] because it may interconnect more than two nodes. As an example, consider instruction 10 in the control flow graph shown in Figure 7.4(a). Because this instruction reads from live ranges B and H and writes to live range G, in the *live range usage graph* shown in Figure 7.4(b), an edge is drawn between the three nodes that represent these three live ranges. Each edge in this graph was assigned a weight to reflect the number of times each corresponding instruction is likely to execute. The edge weights[2] in the graph are given by the numbers in parenthesises. For example, basic block #4 is estimated to execute 100 times, and thus the edge corresponding to instruction 10 is assigned a weight of 100. Basic block #3, however, is estimated to execute only 10 times, and the edges corresponding to instructions 5 and 7

[2] As explained in Section A.1.1, these weights are derived from profiling the execution of the application on a dual-cluster processor.

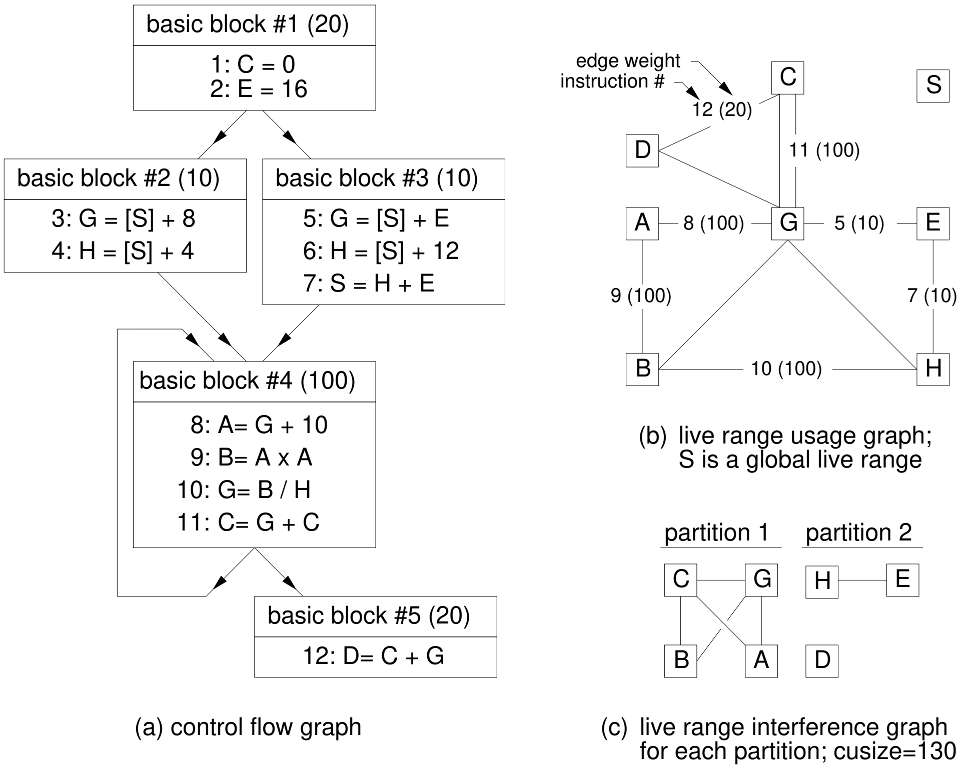


Figure 7.4: Example for discussion of min-cut scheduler.

are assigned a weight of 10. Note that the (hyper-)edge for instruction 5 does not connect to the node for live range S because S is assumed to be a candidate for a global register. In general, edge segments that correspond to reading of a global register are omitted as is the case for instructions 3 to 6. If an instruction writes a live range that is allocated to a global register, the segment that corresponds to the writing is also omitted. Thus, for instruction 7, its edge connects the nodes for live ranges E and H but not that for live range S.

With this graphical abstraction in mind, the problem of minimizing the number of dual-distributed instructions becomes one of finding a legal partitioning of the nodes (i.e., live ranges) that results in the smallest *cutsize*. Through analysis of techniques for bipartitioning graphs, the Fiduccia-Mattheyses graph bipartitioning heuristic [Lengauer 1990] was identified as being most suitable for discovering the partitioning with the smallest cutsize. The cutsize was defined to be the sum of the weights of all *hyper-edges* that cross the boundary between partitions. For example, if live range B is in one partition, and live ranges G and H are in the other, the edge corresponding to instruction 10 would be counted *once* in calculating the cutsize even though two segments of the edge cross the partition. If the partitioning is legal, then the cutsize gives an estimation of the number of instructions that will be distributed to both clusters when the application is run.

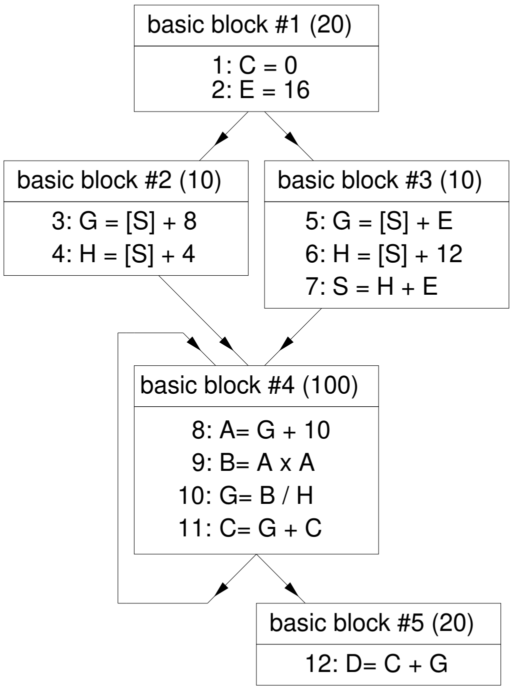
A partition is considered legal if it is possible to map all the live ranges in one partition to the local registers assigned to one cluster of a multicluster processor, and all the live ranges of the other partition to the local registers assigned to the other cluster. That is, there is a one-to-one correspondence between partitions and clusters, and a live range can only be placed in one partition if there is a local register to which it can be assigned. Thus, to judge the legality of a partition, a live range interference graph is built for each partition, and an attempt is made to color it with n colors, where n corresponds to the number of local (architectural) registers assigned to the corresponding cluster. This graph does not include live ranges that are candidates for global registers. If an n coloring is possible, then the partition is legal. Figure 7.4(c) shows the interference graphs for a possible partitioning. This partitioning is legal if each cluster is assigned at least three architectural registers. The min-cut scheduler does not spill live ranges to the other cluster or to memory. This approach reduces the size of the search space and algorithmic complexity.

7.2.4.2 The Global Scheduler

The global scheduler[3] focuses on instruction balance rather than dual-issue minimization. Because instruction balance is an objective, the global scheduler must take into account ordering-induced inter-instruction dependences, and hence, must use a schedule-based approach. The approach taken by the global scheduler is to determine, for each live range L, the cluster to which it should be assigned so as to minimize the run-time instruction-distribution imbalance in the vicinity of every instruction that reads or writes L. A distribution of instructions is said to be *balanced* if the amount of imbalance is never greater than the balance factor β (see the discussion following property 7.1).

Because the objective is to achieve a balanced run-time distribution rather than a balanced static distribution, the order in which live ranges are assigned to clusters is dictated by the number of times each live range is likely to be dynamically read and written. This number is referred to as the *dynamic usage count*. For example, the dynamic usage count for live range A in Figure 7.5(a) is equal to 200 because A is read by instruction 9 and written by instruction 8, both of which are estimated to be executed 100 times. (Each instruction is only counted once in determining the dynamic usage counts.) Figure 7.5(b) gives the dynamic usage counts for the other six live ranges that are candidates for a local register. To order two live ranges that have equal dynamic usage counts, the scheduler ranks a live range L higher if it is read or written by a greater number of static instructions. If the number of static instructions is also equal, as is true for live ranges A and B in the example, the first one encountered in the ranking process

[3] Section A.1.1 presents a more complete description of this scheduler.



(a) control flow graph

	estimated dynamic counts		
	read	write	total
A	100	100	200
B	100	100	200
C	120	120	240
D	0	20	20
E	20	10	30
G	220	120	340
H	110	20	130

(b) estimated usage counts for ranking live ranges

Figure 7.5: Example for discussion of global scheduler.

is given the higher ranking.

The above approach to ranking seeks to give a higher ranking to the live ranges that are most likely to affect the instruction balance at the greatest number of points. As a result, the more highly ranked live ranges will be the first ones assigned to a cluster. In the example considered in this section, assuming that live range B is given a higher ranking than A, and in consideration of the dynamic counts listed under the column headed "total", the compiler would chose a cluster assignment for the live ranges in the following order: G, C, B, A, H, E, and D.

To choose a cluster for a live range L, a cluster preference is determined for each instruction that reads or writes the live range. Once the preferences of all instructions have been compiled, the live range is assigned to the cluster that was preferred by the majority. The preferences are weighted so that the outcome is more heavily influenced by the preferences for instructions with higher execution estimates. The preference for an instruction I is based on the degree of instruction imbalance around the instruction, and the cluster-assignment for L that might permit instruction I to be distributed to only one cluster. The instruction balance around instruction I depends on the live ranges used by I, the live ranges used by the instructions that are near to I in the static schedule, and those live ranges used by the instructions on the control-path preceding the fetching of instruction I. Thus, to precisely quantify the instruction balance around I, it is necessary to evaluate the contribution of every live range in the application, and all possible control paths, a task that is computationally intractable. For this reason, instruction balance is determined on a per-basic-block basis; Section A.1.1 describes how balance is calculated.

When choosing a cluster for a live range L for an instruction I, consideration is given to how the assignment of L to cluster C will affect the instruction balance around I. If the assignment of L to either cluster will not lead to an unbalanced distribution, then either cluster may be chosen. In this case, preference is given to the cluster that ensures the instruction will distribute to only one cluster, if one exists, and otherwise, either no preference is registered, or the scheduler selects the cluster that has been assigned relatively more slave copies than master copies[4]. However, if an unbalanced distribution would result, the scheduler selects the cluster for which the resulting imbalance would be as small as possible.

Note that the cluster assignment for a live range L depends on the cluster preferences of the instructions that use the live range, but these preferences are sensitive to the order in which the live ranges are visited. The source of this sensitivity is the need to take into account the cluster distribution of instructions in the vicinity of an instruction I when the cluster preference is determined by the scheduler for I. Since this distribution can change with the future allocation of any yet unassigned live range, there is no guarantee that the cluster chosen now would be the one chosen in the future. Thus, the global scheduler uses a greedy algorithm to partition the live ranges and to thereby schedule the instructions.

7.2.4.3 The Local Scheduler

The global scheduler attempts to correct instruction imbalance at numerous points in the code. This process is hindered by the instruction balance at one point being directly and indirectly affected by the balance at another point. As a result, this approach can lead to decisions that are neither globally nor locally advantageous. The local scheduler[5], on the other hand, sacrifices this attempt at global optimality for more control over instruction balance in the vicinity of one of the instructions that writes the live range. The scheduler will consider the other instructions that read or write the live range only if the instruction distribution is balanced in the vicinity of the selected instruction.

To determine cluster assignments for the live ranges, the local scheduler begins by sorting the basic blocks according to the number of times the first instruction in each basic block is estimated to be executed. Basic blocks with equal estimates are sorted by the number of (static) instructions in each

[4] By balancing the distribution of master and slave copies, the hope is to balance the requirements for physical registers and transfer buffer entries.

[5] Section A.1.2 presents a more complete description of this scheduler.

block. Then, the basic block having the largest estimate and the greatest number of instructions is removed from the list, and a bottom-up in-order traversal is carried out on the instructions in the block. The purpose of this traversal is to visit each instruction in turn, and if the instruction writes an unassigned live range, choose a cluster for the live range. A cluster is chosen for a live range L, which is written by an instruction I, by first examining the instruction-distribution around the instruction. If the distribution is not balanced, the cluster chosen for L will be the one that reduces the degree of imbalance. However, if the distribution is balanced, then the scheduler determines the cluster that is preferred by the majority of the instructions that read or write L. The scheduler selects cluster C as the preferred cluster for one of these instructions if the assignment of L to C will allow the instruction to be distributed to only one cluster.

Thus, once the bottom-up traversal is completed of a basic block, there will remain no unassigned live ranges among those that are written by the instructions in the basic block. After completing the bottom-up traversal of a basic block, the scheduler removes the next basic block from the list, and carries out a bottom-up traversal on its instructions. This process continues until all basic blocks are visited. As a result of this process, the cluster-assignment for a live range will be determined the first time an instruction is encountered that writes it during the basic-blocks traversals.

For the example of Figure 7.5, the local scheduler will visit the basic blocks in the order 4, 1, 5, 3, and 2, and, as a result, the live ranges will be assigned to clusters in the order C, G, B, A, E, D, and H. In contrast, the global scheduler will visit the live ranges in the order G, C, B, A, H, E, and D.

7.2.4.4 The Local and Global Schedulers, and the Multiflow Compiler

Further insight into the global and local schedulers is provided by comparing their solution to a problem that is also solved by the Multiflow compiler [Lowney 1993]. The problem is to find a partitioning of the intermediate-language instructions that meets the stated objectives. Instruction partitioning is the focus because the physical registers and the functional units of both the multicluster architecture and the Multiflow architecture are distributed among the clusters of the respective machine. (For the Multiflow architecture, the clusters are called arithmetic logic units.) While these two architectures share this common feature, the algorithms used to partition the instructions must be different.

The algorithm used by the Multiflow compiler is based on arranging the instructions into a (static) code schedule. The position that an instruction I is assigned in the schedule is a function of the positions held by the instructions that generate the source operands of instruction I and the availability of the appropriate functional units. The compiler seeks to place instruction I as early as possible in the schedule. Hence, I is assigned to the functional unit that can generate the result as soon as possible after the instruction's input dependences are resolved. Because each functional unit is associated with a single ALU (i.e., cluster), in so allocating instructions to functional units, the instructions are partitioned. This partitioning is based on the availability of the source operands of the instructions and the functional units they require. This basis is possible because the instructions in a Multiflow computer are issued, executed, and completed in a deterministic order.

The approach used by the Multiflow compiler, however, cannot be used by the local and global schedulers, because with this approach, the code schedule is assembled as the partitioning progresses. A complete code schedule is required by the local and global schedulers at the start of the process, because, without one, it is unclear how to determine whether the distribution of instructions is balanced. Because the Multiflow compiler can predict the availability of data and functional units, its partitioning algorithm can be based on time, whereas the algorithm for the local and global schedulers has to be based on instruction sequence. Thus, these schedulers begin by assuming a complete schedule, and manipulate the register allocations to achieve a balanced distribution. At the cost of additional algorithmic complexity, the order of the instructions in the schedule could also be varied, but exploration of this possibility is beyond the scope of this dissertation.

7.2.5 Dynamic Reassignment of Architectural Registers

Although both the local and global schedulers seek to precisely control the run-time instruction balance through (static) instruction scheduling, the scheduling intricacies discussed in Section 7.2.2 preclude an optimal solution, and unpredictable dynamic behavior precludes taking into account all sources of structural hazards. Through the analysis of the multicluster concept that was performed as part of this dissertation, a simple and relatively inexpensive hardware mechanism was developed to reduce the sensitivity of an application's performance to these factors. This mechanism is described in this section.

The mechanism relies on the following property of the multicluster architecture: the determination of the clusters to which an instruction is distributed depends on the architectural registers named by the instruction and the architectural-register-to-cluster assignment in effect when the distribution decision is made. Thus, hardware is used to monitor the relative utilization of both clusters, and, should one be over subscribed, the hardware triggers a reassignment of the architectural registers (i.e., triggers changes to the architectural-register-to-cluster assignment). When a reassignment is triggered, one or more of the *local registers* are reassigned to direct instructions away from the over-subscribed cluster. The local/global designation of architectural registers is not changed.

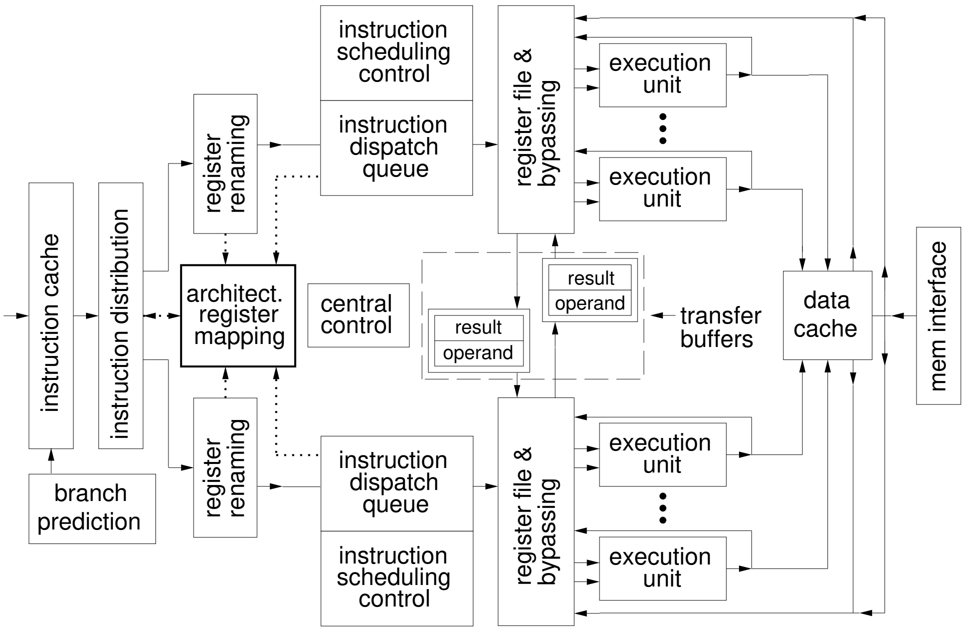


Figure 7.6: The multicluster architecture with the module required to dynamically support reassignment of the architectural-register-to-cluster assignment.

The cluster assignment of a local register R_a may only be changed just before the distribution decision is made for an instruction I_n if I_n names R_a as a destination register. Because instructions are distributed in fetch order, the cluster-assignment changes are also made in fetch order. Thus, instructions earlier than I_n in fetch order will not be affected by the cluster change, but all those following I_n will be affected. If the instruction I_n names R_a as both a source and a destination, the source will use the old cluster assignment, say $C_1 \leftarrow R_a$, while the destination will use the new assignment $C_2 \leftarrow R_a$. In this case, when I_n is subsequently executed, a physical register in cluster C_1 will be read to obtain the old value of R_a and a physical register in cluster C_2 will be written with the new value. This distinction between the physical register R_p containing the old value of R_a and the physical register R_p^n that will contain the new value of R_a is no different than the distinction that is made if dynamic changing of the architectural-register-to-cluster assignment is not supported. The only difference is that, with dynamic reassignment, R_p and R_p^n can now belong to different clusters. For this reason, the conditions under which the physical register R_p can be freed are the same as before, which for precise exceptions is the commitment of instruction I_n.

To record the existing architectural-register-to-cluster assignment, it is proposed that a bit mask be used. In this mask, the value of each bit indicates the cluster assignment for the associated local register. This bit mask is envisioned to be part of the module labeled *architect. register mapping* in Figure 7.6. Because an instruction I can potentially change the cluster assignment of any architectural register, to permit recovery from exceptions, it is necessary to record the value of the bit mask before I is permitted to change the mask. If an exception should then occur when I is executed, the bit mask can be restored to the value it contained before I modified it, and, in so doing, the original cluster assignment is recovered. In practice, however, it may not be necessary to explicitly record the cluster assignment because this information may be implicit in the architectural-to-physical register mapping that is already maintained for each instruction.

The proposed methods for implementing dynamic reassignment of the architectural registers all rely on the above mechanism for changing the cluster assignment of an architectural register. There are four parameters that specify a reassignment implementation. These are: (1) the quantity that is measured for each cluster, (2) the method used to compare the two measurements, (3) the criteria used to determine when a reassignment should be triggered, and (4) the procedure used to reassign the architectural registers once a reassignment has been triggered.

A number of quantities can be measured for each cluster to estimate the utilization of the cluster. Three proposed metrics with reasonable implementation costs are: the number of free dispatch-queue entries, the number of free physical registers, and the instruction flow rate through the dispatch queue. The underlying premise of the first two metrics is that if both clusters have the same number of occupied dispatch-queue entries or in-use physical registers, then the instruction scheduler of each cluster will be able to find enough instructions to issue, and thus, the workload will be balanced across the clusters. This premise is also the basis for the min-cut, local and global schedulers. The objective of the third metric is to gauge more directly the workload balance. This metric measures the difference in the number of instructions, per cycle, that are inserted into a dispatch queue and that are issued out of the dispatch queue. The three metrics are listed in column 1 in Table 7.1. The other columns in this table correspond to the other three design parameters, and these are described below.

To determine whether one cluster is over subscribed while the other is under subscribed, the measured quantities for both clusters are compared, either by taking their difference, or by taking their ratio. For ease of discussion, let the result of the comparison be Δ. If Δ is outside a predetermined range of values, then one of the clusters is considered over subscribed. The two comparison methods are listed in column 2. A reassignment of the architectural registers is triggered if one cluster has been over subscribed for the last n cycles, where $n \geq 1$. Larger values of n correspond to a less greedy implementation since short lived workload imbalances are not acted upon.

When a reassignment is triggered, one or more registers are reassigned. A number of implementations were developed. In the simplest, the value of Δ was evaluated before each instruction I was dispatched

trigger parameters			reassignment procedure
quantity measured (1)	comparison method (2)	criteria (3)	(4)
• busy dispatch-queue entries • busy registers • instruction flow rate	• ratio • difference	• last n cycles, $n \geq 1$	• one register • one register until zero • all registers
one of the above	one of the above	• backward branch	• all registers

Table 7.1: Methods for dynamically changing the architectural-register-to-cluster assignment. A method is defined by choosing from the columns a quantity to be measured, a method of comparison, a criteria for determining when to trigger a reassignment and a procedure used to do the reassigning.

to one or both clusters. If this value was outside the predetermined range and if the register written by I was assigned to the over-subscribed cluster, then this register was reassigned to the under-subscribed cluster. This approach is referred to as "one register" and is listed in Table 7.1 in column 4. In the worst case, however, the one-register approach may lead to every second instruction having its destination register reassigned. To avoid this potential for oscillation, the second implementation was developed. In this implementation, once Δ was outside the allowed range of values, the reassignment of destination registers continued until Δ became zero. At this point, no more registers were reassigned. This approach is referred to as "one register until zero" in the table.

The third, more complicated implementation, differs from the other two in that the hardware responds to a reassignment trigger by reassigning all local registers rather than a single register. This implementation is referred to as "all register" in the table. In this third implementation, when the trigger was raised, flags are set to signify that all registers should be reassigned. However, the actual reassignment for a register R_a did not occur until an instruction I was dispatched that writes R_a. To provide the required functionality, a second bit mask was used. The purpose of this mask is to note whether a register should be reassigned. As instructions are dispatched to the appropriate clusters, the bit corresponding to their destination register is checked. If the bit is set then this bit is cleared and the cluster-assignment bit for the register is flipped, thereby reassigning the architectural register. It is possible, however, that when a reassignment is triggered, some of the registers will not have been reassigned since the last trigger was raised. To avoid assigning all the registers to the same cluster, only pending bits not already set are set. In other words, the new value of the pending bit mask is equal to the complement of the old value.

The proposed methods described so far for reassigning the architectural registers share the common feature of basing the reassignment trigger on measurements made by the hardware. Because the hardware can only track past behavior, any hints of future behavior must be supplied by the software. Implementing such hints was beyond the scope of the research presented in this dissertation. However, a mechanism was devised to evaluate the potential for such hints in a preliminary way. In the simulation framework used in this dissertation, a trigger mechanism was implemented that constrained the reassignment of registers to immediately before the insertion into a dispatch queue of a backwards branch. A backwards branch transfers control to an instruction at a lower program-counter address. As such branches frequently define the backwards edge of a loop, triggering a reassignment on the branch will reassign the registers for the next loop iteration. In this method, a comparison of the measured quantities is made over the last n cycles to determine whether to trigger a reassignment. The parameters for this method are specified in the bottom row of Table 7.1. The discussion in Section 7.3.3 describes an implementation of this method as well as one using some of the other options presented in the table.

7.3 Evaluation of Instruction Schedulers for Multicluster Architecture

Simulations using a number of the SPEC92 benchmarks were performed for this dissertation to evaluate the performance impact of the min-cut, global and local schedulers. The goals of the evaluation were to validate the assumptions on which the schedulers are based, to investigate the performance impact of various design parameters and thereby refine the algorithms on which the schedulers are based, and to attempt to quantify the difficulties that out of order issue imposes on the static generation of code. The methodology used in this investigation is presented in the next section. Then, Section 7.3.2 considers the use of the schedulers with a multicluster architecture in which the architectural-register-to-cluster assignment is hard-wired, and Section 7.3.3 considers the use of the mechanism discussed in Section 7.2.5 for dynamically reassigning the architectural registers.

7.3.1 Methodology

The min-cut, local and global schedulers were developed within the five-step framework that was developed as part of this dissertation; this framework is described in Section 7.2.3. The schedulers were implemented using ATOM [Srivastava 1994], an object code instrumentation package, and standard DEC OSF/1 compilers. While this approach prohibited the use of compilation techniques (e.g., loop unrolling) not supported by the DEC OSF/1 compilers, but which might improve the performance of the multicluster architecture, it simplified the implementation of the schedulers and permitted the fundamental scheduling problems to be the focus of the work.

Because the DEC OSF/1 compilers were used as a front end, there was no need to implement steps one or two of the scheduling framework. To implement steps three to five of the scheduling framework, the following four-part procedure was used.

1. The *native binary* (i.e., the object code) generated by the DEC OSF/1 compilers was analyzed using ATOM to discover the data and control dependences between instructions, and the live ranges these instructions read and write.

2. The live ranges were partitioned and assigned to the architectural registers using the min-cut, global, or local scheduler. The object code generated by the schedulers is referred to as the *rescheduled binary*. For this step, a mapping between the architectural registers and clusters is required. The mapping that was used was determined through an analysis of early simulation results. This analysis indicated that a good mapping was to assign even-numbered registers to cluster C_1 and odd-numbered registers to cluster C_2; this analysis is discussed in Section 7.3.2 (see page 155). Regarding the local versus global designation of registers, since applications tend to read the stack and global pointers much more frequently than they are written, the two registers used to hold these values were assumed to be global, whereas, all other architectural registers were assumed to be local.

3. The rescheduled binary was then instrumented using ATOM and linked with the multicluster simulator. This process is described in Section 3.5. Instrumentation took into account the new register assignments for each instruction and any code required to spill live ranges.

4. The combined application-simulator was then run and statistics were recorded. The number of (simulated) clock cycles required to execute a benchmark was used as a measure of its run-time performance.

To evaluate the impact of rescheduling, the performance obtained from the rescheduled binaries was compared to that obtained when the native binary was executed on a dual-cluster processor and on a single-cluster processor. For this comparison, the single-cluster processor was configured with the same number of resources as the entire dual-cluster processor. If factors that affect the run-time behavior of an application, such as cache misses and branch mispredictions, are ignored, it is always true that $N_{multi} \geq N_{single}$, where N is the number of (simulated) clock cycles required to execute an application. This relationship exists because it is highly probable that some instructions will be dual distributed and dual issued in a multicluster processor, and thus, it is likely that this duplication of instructions will

processor	all	integer	floating point				loads and stores	control flow
	max # of operations that can be issued in a cycle							
			all	divide	multiply	other		
single	8	8	8	4	4	4	4	4
dual, per cluster	4	4	4	2	2	2	2	2

Table 7.2: Instruction-issue rules for the single-cluster processor and for each cluster of the dual-cluster processor.

increase the number of clock cycles required to complete the application. Because performance is a product of the number of clock cycles and the period of the clock T, a multicluster processor will perform as well as or better than a single-cluster processor if the clock period of the multicluster processor T_{multi} is less than or equal to $\frac{N_{single}}{N_{multi}} \times T_{single}$.

Although the evaluation was done for both four-way and eight-way issue processors, the results presented here are only for an eight-way issue processor because these more clearly show the important trends. The single-cluster, eight-way issue processor was configured with 128 physical registers, 128 dispatch-queue entries, an inverted MSHR, and a memory interface with a 16 cycle fetch latency and unlimited bandwidth. The issue rules for this processor are given in the first row of Table 7.2. The multicluster processor used for the performance evaluations was configured with two clusters each capable of issuing up to four instructions per cycle. Each cluster had 64 physical registers, 64 dispatch-queue entries, and the instruction issue rules given in the second row of Table 7.2. The multicluster processor had the same MSHR organization and memory interface as the single-cluster processor, and both enforced precise exceptions.

7.3.2 Design Implications Without Dynamic Register Re-assignment

This section presents and discusses the results of the work done using the four-part procedure described in Section 7.3.1, to investigate the effectiveness of the three proposed schedulers in improving the performance of applications running on a dual-cluster processor with a hard-wired architectural-register-to-cluster assignment. A number of trends were observed in this investigation, and these are discussed in this section in the context of six benchmarks.

A common (and expected) trend is that more clock cycles are required to execute a benchmark on the dual-cluster processor than on the single-cluster processor. This increase is attributable to an increase in the number of stalls of the instruction-fetch stream, a reduction in the number of instructions issued per cycle, an increase in the instruction-issue disorder, and a slight increase in the data-cache miss rate. A

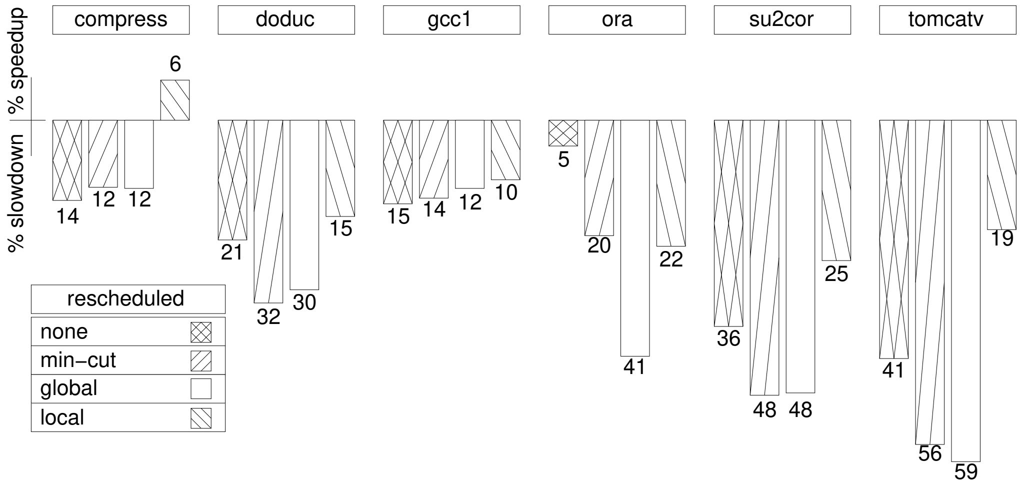


Figure 7.7: The performance ratios obtained with and without rescheduling the benchmarks.

useful metric is $\frac{N_{dual}}{N_{single}}$, where N_{single} is the number of (simulated) clock cycles required to execute the native binary of a benchmark on the single-cluster processor, while N_{dual} is the number of (simulated) clock cycles required to execute either the native binary or one of the three rescheduled binaries on the dual-cluster processor. This performance ratio is said to indicate a *speedup* if it is less than one, and a *slowdown* if it is greater than one.

The performance ratios for each benchmark are presented in Figure 7.7, with one group of four bars for each benchmark, and each bar corresponding to one of the four binaries that were simulated. The bars indicate the percentage speedup (slowdown) with the number located above (below) each bar giving the speedup (slowdown) percentage explicitly. The speedup/slowdown percentage is equal to $100(1 - \frac{N_{dual}}{N_{single}})$. Comparison of all 24 data points in the figure indicates that in general, the benchmarks incurred a slowdown of between 5% and 59%, with 26% being the average. Furthermore, comparison of the bars of the four methods for generating the program binaries indicate the following:

- With the exception of *ora*, the use of the local scheduler gives the best performance ratio. Furthermore, in one case (i.e., *compress*) the ratio is greater than one.
- With the exception of *ora*, the use of the min-cut or the global scheduler gives similar performance ratios.
- For four of the benchmarks, the use of either the min-cut or the global scheduler gives much worse performance ratios than those given by the use of no scheduler.

These trends can be understood by examining the run-time workload distribution.

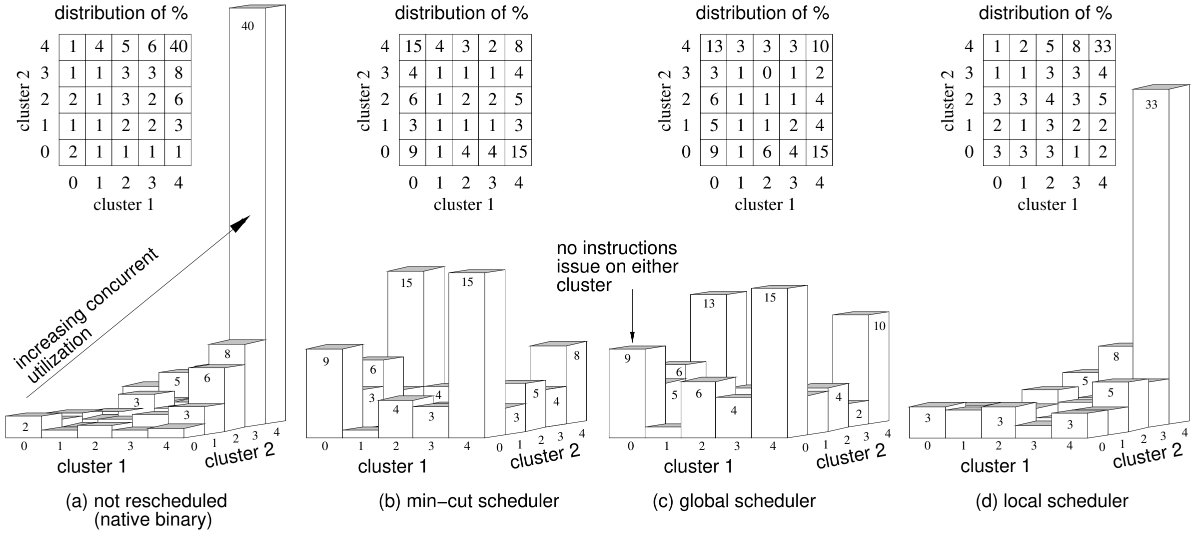


Figure 7.8: The correlated number of instructions issued by each cluster in the same cycle for *su2cor* with and without rescheduling.

A measure of the dynamic workload distribution is the per-cycle correlation between the number of instructions issued by one cluster and the number issued by the other. Figure 7.8 presents instruction-issue correlation information for four runs of *su2cor*. The figure shows the percentage of the clock cycles in which cluster C_1 issued N_1 instructions (the left to right axis) and cluster C_2 issued N_2 instructions (the front to back axis), with $0 \leq N_1, N_2 \leq 4$. For example, in Figure 7.8(a), coordinate $(N_1, N_2) = (0, 0)$ indicates that, with no rescheduling, for 2% of the clock cycles neither cluster issued any instructions, while coordinate $(4, 4)$ indicates that for 40% of the cycles both clusters issued the maximum number (i.e., 4 instructions per cluster). The path traced from $(0, 0)$ to $(4, 4)$ corresponds to increasing concurrent utilization of the issue bandwidth of both clusters. Points farther from this line, say $(1, 4)$, represent increasingly less balanced workload distributions and a lower degree of concurrent utilization of the hardware.

Insight into the relatively poor performance obtained using the min-cut and global schedulers is provided by comparing Figures 7.8(a), (b) and (c). Consider first Figure 7.8(a). The behavior here is dominated by the concurrent issuing of more than six instructions per cycle by both clusters, and in particular, by the behavior at coordinate $(4, 4)$. This observation suggests a high degree of concurrent utilization of both clusters. In contrast, the behaviors of the min-cut and global schedulers are dominated by low degrees of concurrent utilization. In particular, coordinate $(0, 4)$ indicates that for ≈15% of the clock cycles, when cluster C_2 issued four instructions in a cycle, cluster C_1 issued none; coordinate $(4, 0)$ indicates that the same phenomenon occurred with the roles of clusters C_1 and C_2 reversed. Thus, for

$\approx$30% of the clock cycles, one cluster issued four instructions while the other was idle. When the native binary was used, the $(0, 4)$ and $(4, 0)$ issue scenarios accounted for 2% of the clock cycles. Clearly, the native binary is better able to take advantage of the hardware resources of the dual-cluster processor, and thus, achieves better performance.

Consider now Figures 7.8(a) and 7.8(d). Observe that both have similar performance distributions, and that these distributions suggest a high degree of concurrent utilization. The frequency of eight-way issue, however, is 7% less when the local scheduler is used, which is partially due to a decrease in the number of dual-issued instructions (64% to 40%). Moreover, with the local scheduler, instructions were issued more in order, with the effect that the number of deadlock-avoidance exceptions (see Section 6.1.1.2) dropped by a factor of 1000 (from 1 per 1000 to 1 per 1,000,000 committed instructions). These run-time behavior changes contributed to the local scheduler generating a binary that required 8% (=$100 - 100\frac{N_{local}/N_{single}}{N_{none}/N_{single}} = 100 - 100\frac{1.25}{1.36}$) fewer clock cycles to run than the native binary. Similar changes are exhibited by the use of the local scheduler with the other benchmarks, thus suggesting that the performance benefit obtained with the local scheduler is due in part to better utilization of hardware resources.

It is nonetheless surprising to note that even with no rescheduling, the benchmarks achieve reasonable utilization of the hardware, a fact that is due to an unintentional balancing of the workload distribution. This balanced distribution is due to artifacts of the register allocation strategy used by the DEC OSF/1 compilers. This strategy categorizes the architectural registers into a number of distinct groups, with each group containing an equal number of even-numbered and odd-numbered registers. These groups are used for specific tasks such as holding temporary values or computing addresses. As a result, similar tasks are more typically performed using the same set of registers, and, because half of these registers are assigned to each cluster, the work tends to be distributed to both clusters (see point 2 on page 151). The fairness of this distribution tends to be better for procedures with smaller numbers of simultaneously active live ranges, which is a by-product of the procedure calling convention. This convention requires that certain groups of registers be saved on entry into a procedure and restored on exit from the procedure, while registers in other groups are assumed to be dead on entry and exit and thus don't need to be saved. The register allocators favor the registers that don't need to be saved, because these registers won't contain an active live range at procedure boundaries. Therefore, some spills of live ranges to memory can be avoided. A consequence of favoring a subset of registers in this way is that the allocator heavily uses the registers in the subset. Since half of these registers are assumed to be assigned to each cluster, it is highly likely that a fair instruction distribution will occur at run time.

The preceding discussion leads directly to consideration of the underlying premise of the multicluster architecture that the cycle time of the processor may be decreased by reducing the size and complexity of key hardware components such as the register file. The performance ratios presented in Figure 7.7 indicate that with the use of the local scheduler the largest slowdown is 25%. To compensate for the increase in clock cycles that this slowdown represents, the dual-cluster processor would have to use a processor clock with a period 20% smaller ($= 100 - 100\frac{T_{dual}}{T_{single}} = 100 - 100\frac{N_{single}}{N_{dual}} = 100 - 100\frac{N_{single}}{1.25 \times N_{single}}$). To assess the viability of a 20% reduction, assume that the cycle times of the single-cluster and dual-cluster processors scale with the cycle time of their register files. The register files for the eight-way issue single-cluster processor have 128 registers and are assumed to have 16 read ports, and 8 write ports (because the processor can issue 8 instructions per cycle). Using the register file cycle times given in Figure 4.6, the register file cycle time for the single-cluster processor is 4.75 nanoseconds (this value corresponds to the data point on the integer-register-timing curve in Figure 4.6(b) for 128 registers). On the other hand, the register files for the dual-cluster processor have 64 registers, and are assumed to have 8 read ports, and 4 write ports (because each cluster can issue 4 instructions per cycle). Thus, the register file cycle time is 3.67 nanoseconds, corresponding to the data point on the integer register timing curve in Figure 4.6(a) for 64 registers. Using these two cycle times, the cycle time of the dual-cluster processor is estimated to be 23% smaller ($1 - \frac{T_{dual}}{T_{single}} = 1 - \frac{3.67}{4.75}$). Because only a 20% reduction in the clock period is required to offset the worst-case 25% slowdown, the dual-cluster processor with a 3.67 nanosecond clock could potentially deliver 3% better performance for the six benchmarks than the single-cluster processor with a 4.75 nanosecond clock.

7.3.3 Design Implications of Dynamic Register Re-assignment

In this second part of the investigation, the performance impact of the three schedulers was evaluated using a dual-cluster processor which supported dynamic reassignment of the architectural registers. The objective of this reassignment mechanism is to decrease the performance sensitivity of the processor to (static) instruction scheduling. Table 7.1 lists the proposed methods of implementing this mechanism, all of which are based on the idea of dynamically reassigning the local registers to clusters when hardware monitors detect that one cluster is over subscribed. In total, 25 implementations were investigated. For each implementation, simulations were carried out to evaluate the performance sensitivity of relevant design parameters and the performance obtained when used in conjunction with several configurations of the min-cut, local, and global schedulers. Overall, approximately 200 simulations were performed for each issue width of the processor for each of seven benchmarks, and the following conclusions can be drawn from the approximate total of 2800 simulations:

1. The local scheduler with no hardware for reassigning the architectural registers almost always performs better than all other configurations.

2. The local scheduler along with the reassignment implementations based on detecting backward branches performs nearly as well as with no reassignment hardware.

3. The global and min-cut schedulers performs similarly and poorly for all configurations.

4. The performance was fairly insensitive to the quantities used to control the reassignment, for reasonable values.

To illustrate these trends, simulation results are presented below for two of the many implementations that were considered.

Both implementations (i.e., schemes) use two metrics to estimate whether one cluster is over subscribed while the other is under subscribed. These metrics are: (1) the relative number of in-use dispatch-queue entries (N_d), and (2) the relative number of in-use physical registers (N_r). The values of N_d and N_r are compared by computing the ratio of the per-cluster values of each metric. That is, $\Delta_{dis} = \frac{N_d^1}{N_d^2}$ and $\Delta_{reg} = \frac{N_r^1}{N_r^2}$, where N_d^1 and N_r^1 correspond to cluster C_1, and N_d^2 and N_r^2 correspond to cluster C_2.

A register reassignment is triggered if either ratio indicates that one cluster is over subscribed by a factor of 2 for 4 clock cycles. In other words, a reassignment is triggered if one of the following is true for 4 successive clock cycles: $(\Delta_{dis}\ or\ \Delta_{reg}) > 2$ or $(\Delta_{dis}\ or\ \Delta_{reg}) < \frac{1}{2}$. When a reassignment is triggered, the *first scheme* continues to reassign architectural registers until $\frac{1}{2} \leq \Delta_{dis}, \Delta_{reg} \leq 2$ for 4 successive clock cycles. With the *second scheme*, when a reassignment is triggered, registers are not reassigned until a backwards branch is to be inserted into a dispatch queue. Once triggered, reassignment of the local registers continues until either all registers have been reassigned once (i.e., only one reassignment per register is allowed) or another reassignment is triggered.

Table 7.3 presents clock-cycle ratios that correspond to the use of these two reassignment schemes, and/or the use of the local or global schedulers to generate the binary. These performance ratios are equal to $\frac{N_{baseline}}{N_{modified}}$, where $N_{baseline}$ is the number of clock cycles required to execute the native binary of a benchmark on the baseline processor, and $N_{modified}$ is the number of clock cycles required when one of the reassignment schemes and/or one of the schedulers are used. The baseline processor is an eight-way issue dual-cluster processor with a hard-wired architectural-register-to-cluster assignment.

bench-mark (1)	clock cycles baseline processor (2)	clock-cycle ratio relative to baseline processor							
		hard-wired		first scheme			second scheme		
		local (3)	global (4)	native (5)	local (6)	global (7)	native (8)	local (9)	global (10)
compress	40.5	1.22	1.02	1.02	1.14	0.98	1.00	1.21	1.02
doduc	69.0	1.04	0.93	1.06	1.00	1.02	0.99	1.03	0.94
gcc1	8.9	1.04	1.02	1.01	1.00	1.00	1.00	1.04	1.02
ora	96.9	0.86	0.74	1.02	0.88	0.73	0.99	0.86	0.74
su2cor	110.1	1.09	0.92	1.07	1.02	1.00	1.00	1.10	0.90
tomcatv	226.3	1.19	0.89	1.13	1.11	1.08	1.01	1.20	0.87

Table 7.3: The performance ratios relative to the baseline processor obtained with the use of the local or global schedulers, and/or the optional use of the first or second scheme for reassigning local registers. Column 2 gives the number of (simulated) clock cycles N in millions for the baseline processor, while the other columns give the ratio of $\frac{N_{baseline}}{N_{modified}}$. Columns 3 and 4 give the ratios when neither reassignment scheme was used. Columns 5 and 8 give the ratios for the native binaries, while columns 6 and 9, and 7 and 10 give the ratios for the binaries generated by the local and global schedulers, respectively.

Consideration of the performance values in the table supports the trends noted in points 1 to 3 derived from the 2800 simulations. Observe that: (1) when the local scheduler is used, the performance obtained with a hard-wired architectural-register-to-cluster assignment is better in most cases than that obtained when the first reassignment scheme is used, and similar to that obtained when the second scheme is used, and (2) the use of the global scheduler results in consistently poorer performance.

Observe also that the use of the local scheduler with the first scheme tends to result in poorer performance than the use of the local scheduler with the second scheme, while the use of the global scheduler and the first scheme tends to result in slightly better performance. The combined use of the local scheduler and the second scheme tends to provide larger performance improvements because this scheduler tends to generate more balanced instruction distributions, and the second scheme has less impact on the distribution. As noted above, the second scheme triggers a register reassignment only on the backwards edge of a branch, and, more importantly, it continues to reassign registers until all registers have been reassigned once. Thus, it is likely that after a reassignment trigger, the entire architectural-register-to-cluster map will be inverted, thereby resulting in the same assignment as was assumed during static scheduling.

On the other hand, in the first scheme, each register can be reassigned many times, and the reassignment process is likely to be terminated long before the assignment of each register is changed. Since there is a much lower likelihood that the entire map will be inverted after a trigger, it is more likely that the balance generated by the local scheduler will be perturbed. The global scheduler, in contrast, tends to generate an unbalanced instruction distribution, and thus, the skewed assignment generated by the first scheme has a more positive effect on the balance obtained at run time.

The performance results were not sensitive to the quantities used to measure the relative utilization of both clusters or to the method used to compare the measurements, as long as reasonable threshold values were used. This observation suggests that while the concept of "over/under subscription" appears to be relatively simple, quantification of the degree of over/under subscription is difficult, in part because the hardware is forced to use past behavior to predict future behavior. For example, suppose at some point in the execution of an application, one cluster may be idle due to an instruction-stream stall, a stall induced by the unavailability of a floating-point register in the other cluster. The lack of a free register might be, say, the result of a large number of already-completed instructions waiting for an instruction I to complete so they can all commit. When I does complete, these waiting instructions will be able to commit within a few cycles, thereby shifting the bottleneck to another resource. During the stall, the hardware may try to end the stall earlier by reassigning the architectural registers, and, in so doing, redirecting some instructions to the less busy cluster. But, the hardware cannot predict whether such an action will help, have no effect, or make matters worse. Moreover, the redirection is not instantaneous because the register-to-cluster map can only be changed incrementally. Thus, it might have been more advantageous to wait out the stall rather than trying to shorten its lifetime.

This example illustrates two problems that are common to both static and dynamic approaches to achieving a balanced utilization of both clusters. First, the decisions made to correct a workload imbalance concern variables that are only indirectly related to the workload balance. In other words, the focus of these decisions is balancing the distribution of instructions, but there is no direct relationship between distribution and cluster utilization. Second, these two approaches cannot make anything but locally-advantageous decisions. For both, the non-determinism induced by out of order instruction issue and by the complex interaction between hardware resources induced by the instructions are significant barriers to making globally-advantageous decisions.

7.4 Discussion of Non-optimality of Instruction Scheduling

This section reconsiders the objectives of (static) instruction scheduling for dynamically-scheduled processors that were discussed in Section 7.1.2 in light of the understanding gained in developing instruction-scheduling algorithms for the multicluster architecture. In particular, this section examines the reasons why a compiler cannot statically predict the run-time effect of scheduling decisions, and the resulting implications.

In assessing the relative worth of scheduling options, the compiler must evaluate whether each schedule will result in a balanced distribution of hardware-resource requirements in the instruction-stream at run time. To perform this evaluation, the conditions under which a stream is considered balanced must be known at compile time. Dynamically-scheduled processors, however, do not meet this requirement because the availability of hardware resources cannot be statically predicted. Rather, their availability depends on the order in which instructions are issued, completed and committed. To illustrate a scenario in which the conditions are unpredictable, consider the MIPS R10000 processor.

The R10000 processor, as discussed in Section 2.1.4, can support 32 in-flight instructions, and has three 16-entry queues to hold the instructions that are waiting to be issued, one for integer instructions, one for floating point instructions, and one for load and store instructions. Instructions are inserted into these queues in fetch order, with the result that the instruction-stream will stall if there is no available entry in the appropriate queue for the next-to-be-inserted instruction. To prevent such stalls from occurring, the instruction stream must contain a mixture of the instruction types that matches the availability of the dispatch queue entries. Thus, for example, if the address queue contains a number of memory instructions, most of which are waiting for a cache miss to be resolved, the number of loads and stores in the fetch stream should be reduced until these misses are resolved. If the number is not reduced, it is possible that the hardware will stall the instruction stream even though there is sufficient support to process the integer and floating-point instructions that are later in the fetch stream than the stall-inducing memory instruction. However, adapting the composition of the fetch stream is beyond the ability of the compiler because: (1) it cannot statically predict the availability of the dispatch-queue entries, and (2) the same static sequence of instructions, say those in a basic block, may need to be scheduled with different balance criteria to allow for the number of free queue entries being different each time these instructions are fetched.

This second problem might be avoided using a scheme similar to the scheme proposed in Section 7.3.3 for dynamically reassigning the architectural registers to the clusters of a multicluster processor. The purpose of this reassignment was to rebalance the workload to take into account the current availability of dispatch-queue entries and physical registers. A more general implementation of this approach is to have the compiler generate a number of code schedules, each with different resource requirements, and to provide the hardware with a mechanism for selecting from the available schedules the one whose resource requirements best match the resources that are available. It is, however, unclear that such a scheme would perform any better than did the register re-mapper scheme for the multicluster architecture. The

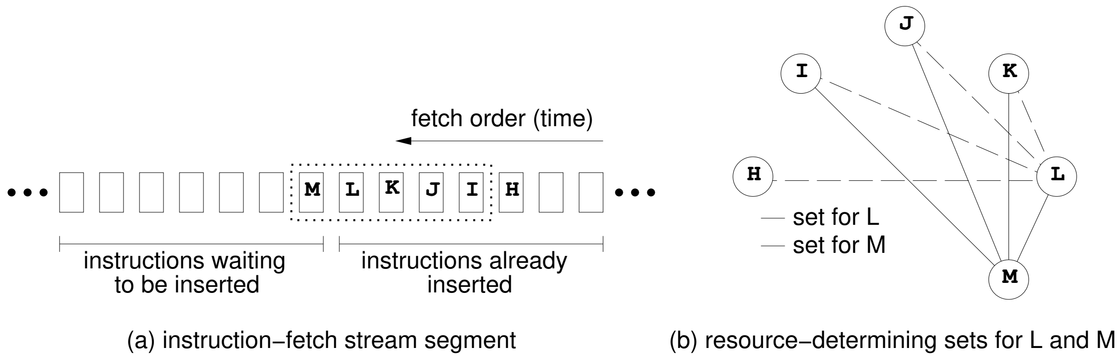


Figure 7.9: A segment of the instruction-fetch stream in which some instructions have already been inserted into the dispatch buffer and others are waiting to be inserted. Also shown is the set of instructions that determine the insertion conditions for instructions M and N.

problem with both of these workload-rebalancing schemes is that the hardware is attempting to improve the near-term performance of the application, but it cannot determine what the long-term effects will be. For example, if the dispatch queue for integer operations in the R10000 processor is over subscribed, the hardware could select a code schedule that trades integer operations for loads, but if it did, the cache miss rate may increase, thereby negating the advantage of the new schedule.

This example illustrates a problem that is common to both static and dynamic approaches to achieving a balanced distribution of hardware-resource requirements for a dynamically-scheduled processor. To aid in the discussion of this problem, consider the instruction-fetch-stream segment shown in Figure 7.9(a). In this figure, the boxes represent instructions, with those towards the left appearing later in the fetch order; thus time is shown increasing from right to left. In addition, instruction M and those to the left of it are assumed to be waiting insertion into the dispatch buffer while the instructions to the right of instruction M have already been inserted. Consider now instruction M. At the point in time that the hardware considers inserting this instruction into the dispatch buffer, the resource requirements of the instructions preceding it in the instruction-fetch stream determine whether the insertion can proceed or whether a stall is warranted. In the figure, it is assumed that no more than four earlier instructions are required to determine the availability of the resources. These four earlier instructions for instruction M are shown enclosed by a box with dotted edges along with instruction M.

If a stall is warranted for the scenario pictured in the figure, because these four instructions determine the availability of the resources for instruction M, it is possible that by re-arranging them, no stall would be necessary. In other words, whether a stall is mandated is determined by the instructions in the vicinity of instruction M. Furthermore, it might be possible to avoid a stall by optimizing the resource requirements of these instructions. However, it is also possible that in trying to avoid a stall for instruction

M, the characteristics of the stream will be changed so a stall would have been warranted for instruction L when it was next in line for insertion. Note that the insertion of instruction L is affected by the resource requirements of the four instructions that precede it. And since instructions K, J, and I affect the insertion of both instructions M and L, a dependence exists between the re-arrangements done to optimize the insertion of each of instructions M and L.

With the above discussion in mind, the aforementioned problem common to both static and dynamic approaches is now noted: to ensure that instruction M is inserted without a stall, the resource requirements of those instructions immediately preceding it should be balanced, but optimizations made to ensure that their resource requirements are balanced can have global consequences. Thus, the generation of an entirely balanced instruction-fetch stream requires solving simultaneously many optimization sub-problems, each of which involves only a subset of the instructions, but whose solution may affect the solutions to other sub-problems.

To more rigorously define the problem, reconsider instruction M in Figure 7.9(a). At the point in time that the hardware considers inserting this instruction into the dispatch buffer, the resource requirements of the instructions preceding it in the instruction-fetch stream determine whether the insertion can proceed or whether a stall is warranted. Let $\mathcal{R}_M(t)$ be the set of instructions that influence this determination for instruction M. Because the availability of resources depends on time, the instruction composition of this set is a function of time (t). Moreover, the composition of the set $\mathcal{R}_M(t)$ is not deterministic for two reasons. First, the issue order of instructions at run-time is not deterministic, and thus resource availability is also not deterministic. Second, resource availability depends on non-deterministic events like data-cache misses, branch mispredictions, and exceptions. Thus, the instructions in the set $\mathcal{R}_M(t)$ need not occur in sequence in the instruction fetch stream.

For each static instruction i in a program binary, there is a corresponding set $\mathcal{R}_i(t)$. The members of these sets are the static instructions in the binary. An alternate representation of these sets is a collection of graphs $\mathcal{G}(t)$, with one such graph for each set. The graph $\mathcal{G}_i(t)$ for instruction i consists of a set of vertices $\mathcal{V}_i$, one corresponding to i, and one corresponding to each instruction in $\mathcal{R}_i(t)$. The edges in the graph, denoted $\mathcal{E}_i$, interconnect the vertex corresponding to i with the vertices corresponding to the instructions in $\mathcal{R}_i(t)$. Thus, if v_j is the vertex corresponding to an instruction j, and $e_{i,j}$ is the edge connecting the vertices corresponding to instructions i and j, then $\mathcal{G}_i(t) = (\mathcal{V}_i(t)\ ,\ \mathcal{E}_i(t)) = (\{v_i\,, v_j\}\ ,\ \{e_{i,j}\})\ ,\ j \in \mathcal{R}_i(t)$. Figure 7.9(b) shows the graphs $\mathcal{G}_L(t_1)$ and $\mathcal{G}_M(t_1)$ for instructions L and M under the assumption that at time t_1 only the preceding four instructions in the instruction stream are required to fully specify resource availability.

Since all the graphs $\mathcal{G}(t)$ in the collection are built from the vertices corresponding to the instructions in the binary program, each graph $\mathcal{G}(t)$ can be viewed as a sub-graph of a composite graph $\mathcal{C}$. In this composite graph $\mathcal{C}$, there is a vertex for each instruction i in the program binary. Furthermore, graph $\mathcal{C}$ is a *complete graph* [Gibbons 1985]. That is, every pair of vertices is interconnected by an edge since each instruction can potentially affect the others.

With the graph abstraction in mind, the problem of generating an entirely balanced instruction-fetch stream is one of first determining for each static instruction i its graph $\mathcal{G}_i(t)$, and then optimizing the instructions in this graph to meet the resource balance objective. Existing techniques cannot be used to solve this problem owing to the requirement that time be taken into account at the same time as inter-instruction dependences. However, existing techniques are sufficient for existing processors that are able to maintain a large enough pool of waiting-to-issue instructions that the effects of short-lived stalls can be hidden. Therefore, for such processors, the performance impact of assuming the processor is statically scheduled during compilation is likely minor. Although there are no known studies on the performance impact of static instruction scheduling for dynamically-scheduled processors, analysis of the architectural frameworks suggests that the MIPS R10000 and HP PA-8000 processors tolerate stalls well while the PowerPC 604 and 620 processors do not. As in the case of the PowerPC 604 and 620 processors, a multicluster processor is also less tolerant of stalls. For these less tolerant processors, maintaining a larger enough pool of waiting-to-issue instructions is more difficult owing to the use of decentralized dispatch buffers that can issue instructions to a more restrictive set of the functional units.

In conclusion, the hypothesized goal of instruction scheduling for dynamically-scheduled processors is to ensure that the instruction-fetch stream is not interrupted by the unavailability of hardware-resources that are required to process these instructions. However, two features of dynamically-scheduled processors impede obtaining this goal. First, it must be possible to statically predict the run-time effect of scheduling decisions so as to assess their relative worth. But, owing to the ability to issue instructions out of fetch order, and thus the inability of predicting the availability of hardware resources, such predictions cannot be accurately made. Second, meeting this goal requires the solution to multiple, non-disjoint, non-deterministic optimization sub-problems. In view of these factors, instruction scheduling for dynamically-scheduled processors must be considered to be insolvable with known techniques. Therefore, further research is warranted.

7.5 Chapter Summary

This chapter has presented an analysis of static instruction scheduling in the context of dynamically-scheduled processors, and has described an application of the results of this analysis to the development of schedulers for the multicluster architecture.

A hypothesis has been put forward that the goal of static instruction scheduling for dynamically-scheduled processors is to ensure that the hardware resource requirements of the instruction-fetch stream at run time are balanced. The premise on which the instruction-scheduling goal is based is as follows: if the size of the waiting-to-issue pool of instructions is sufficiently large and diverse, then the dynamic scheduler will be able to find enough instructions to issue in each cycle to keep instructions flowing through the system at the maximum rate. However, the instruction-scheduling goal can only be partially met owing to the following two features of dynamically-scheduled processors: (1) it is not possible to statically predict the run-time effect of scheduling decisions to assess their relative worth, and (2) meeting the goal requires the solution to multiple, non-disjoint, non-deterministic optimization problems. Nonetheless, the performance impact of failing to meet the scheduling goal is dependent on the ability of the processor of interest to maintain an adequate supply of waiting-to-issue instructions.

For the multicluster architecture, it has been hypothesized that balancing the requirements reduces the likelihood of instruction-fetch-stream interruptions and increases the likelihood of a balanced distribution of the instruction across all clusters. The object of a balanced distribution of instructions is that all clusters simultaneously perform similar amounts of work. However, partitioning of the architectural registers of a multicluster processor introduces a source of instruction-fetch-stream stalls that is especially difficult to address statically. Yet, at the same time, because a multicluster processor is more decentralized, static instruction scheduling has a greater role to play in ensuring that the code takes full advantage of the hardware. Moreover, the partitioning of the architectural registers increases the complexity of the above mentioned optimization problems.

As part of this dissertation, three instructions schedulers were developed with the aim of improving the performance of applications running on a multicluster processor. In addition, a hardware technique was developed to reduce the performance sensitivity of the decisions made by the schedulers. Although these four approaches employed different strategies, they all share two fundamental problems. First, the decisions that are made to correct a workload imbalance are based on variables that are only indirectly related to the workload balance. In other words, the focus of these decisions is balancing the distribution of instructions, but there is no direct relationship between distribution and cluster utilization. Second, these approaches cannot make anything but locally-advantageous decisions. For both approaches, significant barriers to making globally-advantageous decisions are introduced by the non-determinism induced by out of order instruction issue, and by the complex interaction between hardware resources that is induced by the execution of instructions.

Simulations of the execution of six benchmarks on a number of dual-cluster processors were carried out to evaluate the effectiveness of the four hypothesized approaches when they were used individually and in conjunction with each other. A common trend in the results of these evaluations is that the local scheduler was the most effective of the four approaches in reducing the number of clock cycles required to execute the benchmarks. In the worst case, 1.25 times more clock cycles were required to execute each benchmark when an eight-way issue dual-cluster processor was used instead of an eight-way issue single-cluster processor. Under the assumption that the cycle time of the single-cluster and dual-cluster processors scale with the cycle time of the register file, it was estimated that the dual-cluster processor with 64 physical registers per cluster could deliver $\approx$3% better performance than the single-cluster processor with 128 physical registers. While recognizing that this estimate is based on a large number of assumptions, it nonetheless indicates that for the configurations considered, the dual-cluster processor is competitive, and warrants further investigation.

Chapter 8

Conclusions and Future Work

The continuing demand for higher performance is driving the development of superscalar processors that can execute ever larger numbers of instructions in parallel while maintaining the outward appearance of sequential execution. To meet this demand, an increasing number of these processors are dynamically scheduled. However, the performance offered by this implementation framework is provided at the price of significant hardware complexity. This complexity challenges computer architects to develop processor designs that balance hardware complexity and compiler technology against performance targets.

The research described in this dissertation represents a first thorough look at the issues introduced by the hardware complexity. The investigation of the issues is focused in three specific areas. The primary focus is the register file, which plays a key role in defining the functionality required from the other components of a system. A related focus is the other components of the memory system, in addition to the register file, that are used to service data-cache misses. These components are: the lockup-free data cache, the stream buffers, and the interface between these components and the lower levels of the memory hierarchy. The third focus is the non-determinism introduced by dynamic scheduling and its implications on the static generation of code that is well tuned to the processor hardware.

This chapter reviews the main conclusions of this research effort and suggests areas for future work.

8.1 Summary and Conclusions

There are three major parts to this dissertation. In the first part, a description was presented of the system model that was used for the research, and of the process used to develop it. In the second part, an analysis of a centralized (i.e., single-cluster) processor was presented. And, in the third part, a description of a novel decentralized (i.e., multicluster) architecture was presented along with the results and an analysis of the multicluster concept.

8.1.1 The System Model

The research described in this dissertation is based on a software model of a dynamically-scheduled processor. The modeling framework was established through the analysis of: (1) the properties and functional requirements of dynamically-scheduled processors, and (2) the proposed and existing techniques that can be used to implement the required functionality. The results and implications of this analysis were described in Chapters 1 and 2. The model was then defined by examining, in the context of the modeling framework, the features of dynamically-scheduled processors that affect the design of the register file and the memory system. Chapter 3 described this process.

The model was designed to balance the competing objectives of accuracy and generality. Accuracy describes how closely the model emulates the behavior of a real (i.e., actual) system with the same functionality as is embodied in the model. Generality, on the other hand, is a measure of how well the results obtained from the model represent real processors with different implementation characteristics. While a very accurate model is useful for estimating the performance of a very specific point in the vast design space of systems, a less accurate model is useful for discovering performance trends and design relationships at various points in the design space.

The competition between accuracy and generality is due to the central role time plays in determining the behavior of dynamically-scheduled processors. This role is a by-product of executing instructions in parallel. While instructions that are executed in parallel are dependent on each other by virtue of having to share hardware resources, all instructions are dependent on each other by virtue of the order in which instructions must be fetched and committed. Thus, the execution of any instruction can affect the temporal relationship between all future instructions.

In light of the tradeoff between accuracy and generality, and the size of the design space, the system model used in this research was selected to be representative of a region in the design space. The performance trends and design relationships that were identified using the model can be used to select a more limited number of design points at which further investigation is warranted.

8.1.2 Single-cluster Processors

The second part of the dissertation presented a two-part investigation of single-cluster processors. First, Chapter 4 presented an investigation of the design and performance relationships between the register file and the following system aspects: the issue width of the processor and the number of functional units, the size of the dispatch queue, and the exception model. Then, Chapter 5 presented an investigation of the performance relationships involving the components in the data memory hierarchy. These components are: the register file, the lockup-free cache, the stream buffers, and the memory interface to the lower levels in the memory hierarchy. The analysis of the simulation results from these two investigations provided the following observations:

Cycle time: The cycle time of a register file is more strongly dependent on the number of ports than on the number of registers. Under the assumption that the cycle time of the processor scales with the cycle time of the register file, an eight-way issue processor with non-partitioned integer and floating-point register files delivered only a 20% increase in the number of instructions committed per second over the number committed per second with a four-way issue processor. This result suggests that to achieve more significant performance gains, the time constraints imposed by the components likely on the critical timing path of the processor can be reduced by partitioning these components to reduce their size and complexity. These components include the dispatch queue, the register file, and the register renaming unit. Such partitioning is the motivation for the decentralized architecture that was investigated in the third part of this dissertation.

Exception model: The additional register requirements for providing precise exceptions in the modeled processors is relatively small. Moreover, because the register file cycle time is more heavily dependent on the number of register file ports than on the number of registers, and in view of all the other hardware required by a dynamically-scheduled superscalar processor, the hardware cost of supporting precise exceptions is relatively little.

In-flight-miss resources: In terms of the average number of instructions committed per cycle (i.e., the commit IPC), in systems with support for fewer than four in-flight misses, the size of the register file and the bandwidth of the memory interface are not significant design parameters, at least for the benchmarks studied. However, increasing the in-flight-miss support beyond that for four in-flight misses provided an increasingly larger performance differential.

(Physical) registers: The commit IPC delivered by systems was more strongly affected by increasing the number of registers than increasing the number of resources for in-flight misses. However, doubling the number of registers resulted in less than a doubling in the commit IPC, because the functionality offered by the dispatch queue and the lockup-free cache constrained the number of registers that can actually be used.

Dispatch queue: Doubling the size of the dispatch queue resulted in less than a doubling in the commit IPC. Moreover, once a certain number of dispatch-queue entries was reached, a greater proportion of the increase in the number of live registers was attributable to the instructions residing in the dispatch queue. Taken together, these two trends suggest that there exists a small number of dispatch-queue sizes that are most cost-effective for the modeled processors.

Memory bandwidth: The bandwidth of the memory interface had relatively little impact on the commit IPC delivered by all systems with a four-way issue processor, and by systems with an eight-way issue processor and support for at most four in-flight misses. But, for systems with an eight-way issue processor and support for more than four in-flight fetches, the memory bandwidth had a more pronounced impact on the commit IPC that was delivered.

Unenhanced stream buffers: For all but one of the benchmarks, there was a net increase in the commit IPC delivered by all modeled systems when they were modified to include unenhanced stream buffers. The speedup obtained with this modification was most significant for systems with a lockup data cache, the larger register-file sizes, and a bandwidth-unlimited memory interface. The speedup obtained was most strongly affected by the data cache configuration, and least strongly affected by the bandwidth of the memory interface. Larger speedups were delivered by systems with an eight-way issue processor due to the effective compression of time that results from issuing more instructions per cycle.

Enhanced stream buffers: Enhancing the stream buffers with memory-traffic filters and dynamic-stride calculators did not significantly change the delivered commit IPC performance beyond that delivered without these enhancements. This trend indicates, at least for the benchmarks studied, that contention for the interface was not a significant problem, and that the address stream generated by cache misses was dominated by interleaved unit-stride streams. However, in spite of the similar performance of many of the enhanced stream-buffer implementations, the cost of those buffers that have traffic filters is *probably* warranted since the system model does not take into account all possible sources of memory traffic. The exact type of filter would have to be determined when a specific system is being considered.

8.1.3 The Multicluster Architecture

The third part of the dissertation presented a two-part investigation of a novel architecture called the *multicluster architecture* [Jouppi]. In the first part, which is presented in Chapter 6, a detailed description of the architecture was developed, and an analysis of the multicluster concept was performed. The development of the description and the analysis of the concept were done as part of this dissertation. In the second part, which was presented in Chapter 7, the importance of static instruction scheduling for dynamically-scheduled processors was analyzed. From this analysis, scheduling algorithms were developed for the multicluster architecture. The analysis of instruction scheduling and the development of the algorithms were also done as part of this dissertation.

The multicluster architecture is a decentralized, dynamically-scheduled architecture in which the register files, dispatch queue, and functional units of the architecture are distributed across multiple clusters. A mechanism is provided to allow the register file of one cluster to be accessed by instructions being executed on another cluster. This mechanism is based on distributing an instruction to more than one cluster for execution. The multiple distribution of instructions increases the number of clock cycles required to execute these instructions, and reduces the number of instructions that can be simultaneously in execution. However, these negative effects are offset by the reduction in the cycle time of the processor clock that is provided by partitioning the register file and other hardware resources. Consequently, an application may require more clock cycles to execute on a multicluster processor than on a single-cluster processor, but its run time may be smaller.

To counter the above noted two negative effects, the objectives of static instruction scheduling for a multicluster processor are: (1) to minimize the run-time number of multiply-distributed instructions, and (2) to ensure a run-time balanced distribution of the instructions required to perform a task across the clusters so that each cluster is concurrently performing similar amounts of work. Taken together, these two objectives seek to maximize instruction throughput. Since the distribution of instructions is based on the architectural registers named by the instructions, static instruction scheduling is the process by which the instructions are ordered and architectural registers are assigned to the operands and results of the instructions.

Three algorithms were developed to implement this process. To focus on the fundamental scheduling problems, these algorithms were developed with the intent of rescheduling already compiled applications; as discussed in Section 8.3.2, one area for future work is to extend the algorithms for use with the source code of an application. In addition to these three scheduling algorithms, a hardware-based scheme was also developed to reduce the performance sensitivity of statically-made scheduling decisions. Simulations of six benchmarks on a number of dual-cluster processor configurations were performed to evaluate the effectiveness of the four resulting scheduling approaches when they were used individually and in conjunction with each other. Under the assumption that the cycle time of the single-cluster and dual-cluster processors scaled with the cycle time of the register file, it was estimated that the dual-cluster processor with 64 physical registers per cluster could deliver $\approx$3% better performance than the single-cluster processor with 128 physical registers. While recognizing that this estimate is based on a large number of assumptions, it nonetheless indicates that for the configurations considered, the dual-cluster processor is competitive, and warrants further investigation.

Static Instruction Scheduling

Through the analysis of static instruction scheduling performed as part of this dissertation, it was concluded that for dynamically-scheduled processors, the objective of scheduling is to ensure that the hardware resource requirements of the instruction-fetch stream at run time are balanced. This focus must be adopted because the order in which instructions are issued at run time cannot be statically predicted. Thus, the compiler must ignore the internal operation of the processor, and concentrate only on trying to ensure an uninterrupted supply of instructions. Meeting this goal for the multicluster architecture is particularly difficult and is especially important owing to the partitioning of the architectural registers. This partitioning introduces a source of instruction-fetch-stream stalls that is especially difficult to address statically.

Note that the instruction-scheduling goal for dynamically-scheduled processors can only be partially met due to two features of dynamically-scheduled processors. First, it is not possible to statically predict the run-time effect of scheduling decisions so as to assess their relative worth. Second, meeting this goal requires the solution to multiple, non-disjoint optimization problems each of which addresses the concerns of individual instructions in the instruction-fetch stream. In view of these factors, instruction scheduling for dynamically-scheduled processors must be considered to be insolvable with known techniques. Nonetheless, existing techniques may be sufficient for processors that can maintain a sufficiently large number of waiting-to-issue instructions to hide the effects of short-lived stalls of the instruction-fetch stream. Static instruction scheduling is considered further in Section 8.3.3.

8.2 Contributions

The contributions of this thesis are:

- A detailed architectural description, with practical implementation in mind, of important features of dynamically-scheduled processors that affect the design of the register file and of the memory system. The existence of a good model will help reduce the number of research projects whose results are not meaningful due to a failure to consider/control important parameters.

- An analysis of the non-determinism introduced by dynamic scheduling, and the challenges this phenomenon presents to computer architects who must design systems that can be used to full advantage by software.
- An investigation of factors that affect the design of a dynamically-scheduled, from the perspective of an engineer. This investigation explored the complicated relationships that exist between the following system aspects:
  - The dispatch queue,
  - The physical registers,
  - The type of exceptions,
  - The issue width of the processor and the number of functional units,
  - The lockup-free data cache,
  - The stream buffers, and
  - The interface to the lower levels of the memory system.
- An analysis of how various design decisions impact the cycle time of a register file. The conclusions from this analysis motivated the investigation of a novel architecture in which decentralization of key components is employed to reduce the cycle time of the processor. A comprehensive examination was conducted of the software and hardware complexities decentralization introduces in a dynamically-scheduled processor.
- An analysis of the functionality required from miss status holding registers in dynamically-scheduled processors.

8.3 Future Work

In the course of the work that has been described in this dissertation, time limitations prevented full consideration of some topics that warrant further investigation.

8.3.1 Dynamic Scheduling and False Dependences

The hardware of a dynamically-scheduled processor imposes a limit on the maximum number of instructions that can be issued in a cycle. However, the number actually issued is a function of: (1)

	bnez	r_0 , L1	; branch if $r_0 \neq 0$
	move	r_1 , r_2	; $r_2 \leftarrow [r_1]$
L1:	beqz	r_2 , L2	; branch if $r_2 = 0$
			...

(a) without cmov

cmovz	r_1 , r_2 , r_0	; $r_2 \leftarrow [r_1]$ if $r_0 = 0$
beqz	r_2 , L2	; branch if $r_2 = 0$
...		

(b) with cmov

Figure 8.1: Example of replacing a short code sequence with a conditional move (cmov) instruction.

the number of instructions with no unresolved dependences induced by instructions earlier in the fetch order, and (2) the availability of the required execution units. One method for increasing the number of pending instructions is to predict the outcome of branches at the time they are fetched and to allow instructions to be issued before all preceding branches have been issued. A related method is for the compiler to reduce short conditionally-executed code sequences with *conditional or predicated instructions* [Patterson 1995].

A conditional instruction has a condition associated with it that is evaluated when the instruction is executed. If the condition is true, the instruction is executed normally, But if it is false, the execution continues as if the instruction was a no-op, with the exception that when the instruction is issued, the dependence induced on other instructions by the conditionally-generated result is removed.

The most common example of a conditional instruction is the conditional move. This instruction moves a value of one register to another if the condition is true. A situation in which such an instruction might be used is illustrated in Figure 8.1(a). This figure shows a code segment in which the move instruction is executed only if register r_0 is equal to zero. Figure 8.1(b) shows a conditional move instruction that performs the same function as the first two instructions in Figure 8.1(a). When this move instruction is executed, the contents of r_1 will be copied into r_2 only if r_0 is equal to zero. Conditional move instructions are frequently used to replace short code sequences that follow a branch instruction that is likely to be mispredicted. By using a conditional move, the performance penalty incurred by mispredicting the branch is avoided. However, this advantage is offset by two disadvantages.

First, a conditional move may introduce false dependences between instructions in the dispatch buffer of a dynamically-scheduled processor. A false dependence would exist between the conditional move instruction (*cmovz*) and the second branch instruction (*beqz*) in Figure 8.1(b) if r_0 is not equal to zero when the *cmovz* is executed; the false dependence is caused by r_2 being used as a source operand by *beqz*. This false dependence may reduce the number of instructions issued per cycle by the dynamic scheduler because, at any point in time, fewer of the instructions in the dispatch buffer have unresolved dependences. Second, if the move specified by a conditional move instruction is not performed, the rename register that was bound to the destination register of the conditional move could have been made available to another instruction.

Since conditional instructions are developing into an important technique for increasing the intrinsic instruction-level parallelism of applications, it would be topical to examine the performance tradeoffs of conditional instructions when used in dynamically-scheduled processors. In addition, an interesting area of research is mechanisms for predicting whether the condition of a conditional instruction will be evaluated as true when the instruction is executed. An instruction whose condition is predicted to be false may not need to be issued by the dynamic scheduler. Furthermore, if the condition is predicted to be true, the hardware may be able to rename the input and output registers rather than issuing the instruction. Of course, a retry would be required in the case where the prediction was wrong.

8.3.2 The Multicluster Architecture

The simulation results presented for the dual-cluster processors suggest that their performance is competitive with that of the single-cluster processors when one of the three developed static instruction schedulers was used to generate the application binaries. Owing to the limitations of the framework used to implement the scheduler, optimizations that require information derived from the source code of an application were not considered. However, there are a number of promising techniques.

Techniques such as superblock scheduling [Chang 1991], trace scheduling [Lowney 1993], and global scheduling [Rau 1993] might be used to increase the number of instructions that can be jointly scheduled, thus permitting a better estimation of the run-time distribution of the workload. Loop unrolling, which is a part of trace scheduling, is an especially promising technique. Loop unrolling can be used to generate a code schedule in which multiple iterations of a loop were interleaved, with each iteration scheduled to use a separate cluster of a multicluster processor. To further increase the performance of loop unrolling, schemes could be devised to decrease the amount of interaction between the iterations of the loop, and thus, the number of inter-cluster data transfers. One such scheme is to duplicate the code that calculates addresses. A similar scheme, called rematerialization [Briggs 1994], is used in register allocation to recompute a value rather than spilling it to memory. A second scheme is to allocate key variables to global registers so that the variable can be accessed from within each cluster without an inter-cluster data transfer. Software pipelining [Allan 1995] might also be used but further research is required to determine how to apply the technique to processors for which the issue order and resource utilization of instructions cannot be statically predicted.

A second set of techniques might be developed to improve the effectiveness of the hardware mechanism that was developed to permit the dynamic reassignment of the architectural registers to the clusters of a multicluster processor. In particular, the compiler could provide the hardware with hints to indicate when the reassignment could be made. Such a scheme would be more sophisticated than the version investigated in Section 7.3.3 in which the assignment was allowed to change only when the instruction-dispatch hardware detected a backwards branch in the instruction-fetch stream. An even more elaborate scheme would be to permit the compiler to directly specify the architectural-register-to-cluster assignment for each architectural register, including whether the register is a global or local register. This functionality would provide additional flexibility in separating a sequence of instructions into a number of partially-independent threads. Additional hardware, however must be included to allow the mask to be changed in the prescribed manner, and to free physical registers once the register to which they have been bound is declared to be non-global.

8.3.3 Static Instruction Scheduling for Dynamically-Scheduled Processors

The generation of an entirely balanced instruction-fetch stream for a dynamically-scheduled processor, as noted in Section 8.1.3, requires solving simultaneously many non-disjoint, non-deterministic optimization problems. Due to the growing popularity of dynamically-scheduled processors, and of hardware that can be reconfigured as it is used, it would be topical to develop a new class of techniques that can solve the problem of time-dependent graph partitioning. In addition, research is also warranted into the performance impact of unbalanced streams, and whether static-instruction-scheduling techniques for statically-scheduled processors are sufficient for existing dynamically-scheduled processors. Furthermore, it would be of interest to identify the architectural features that render these techniques insufficient, and to investigate the related tradeoff between architectural features and software complexity.

8.3.4 Other Architecture-related Research

Finally, there are a number of other architecture-related topics that warrant further research.

Static versus dynamic scheduling: As noted above, the use of dynamic scheduling complicates the task of statically generating code that is well tuned to the processor's hardware. Irrespective of whether existing static-instruction-scheduling techniques are sufficient for dynamically-scheduled processors, there remains the issue of whether a statically-scheduled processor with a sophisticated compiler can perform better than a dynamically-scheduled processor. A performance comparison of processors built within these two frameworks and with the same hardware capabilities would therefore be of interest.

Branch prediction: The simulation results presented in Chapter 4 for the systems with a large number of registers and dispatch queue suggest that, on average, a significant proportion of the available issue bandwidth was not used by the applications. One factor that affects the ability of the benchmarks to fully utilize the issue bandwidth is the accuracy of the branch predictor. In view of the complexities of building wider-issue processors and the cycle-time impact of increasing the issue width, an in-depth investigation of branch prediction for dynamically-scheduled processors is warranted. Such a study is also warranted since the accuracy of the branch predictor impacts a processor's ability to tolerate cache misses.

Cycle time: The motivation of the multicluster architecture is to decrease the cycle time of the processor by partitioning components on the critical timing paths, and thereby improve the performance beyond what is achievable from a non-partitioned architecture. Given the inability of the simulated benchmarks to fully utilize the resources of a non-partitioned architecture, and the complexities partitioning introduces for static instruction scheduling, a question for further research is whether multithreaded architectures are the key to improving performance beyond what more conventional architectures can achieve.

Appendix A

Instruction Scheduling for the Multicluster Architecture

This appendix continues the discussion of Sections 7.2.4 and 7.3.1 on the (static) instruction schedulers for the multicluster architecture that were developed as part of this dissertation. In particular, Section A.1 describes in more detail the live-range-partitioning algorithms on which the local and global schedulers are based. Section A.2 then describes in more detail how the schedulers were implemented using ATOM [Srivastava 1994].

A.1 The Local and Global Schedulers

As discussed in Section 7.2.3, a five-step approach is proposed for scheduling an application for execution on a dual-cluster processor:

1. The application is compiled into an intermediate-language,
2. The instructions are arranged into a code schedule,
3. The live ranges embodied in the schedule are classified as candidates for either a local register (i.e., an architectural register that is assigned to only one cluster) or a global register (i.e., an architectural register that is assigned to both clusters),
4. The local live ranges are partitioned, and
5. The live ranges are allocated to architectural registers.

The two goals of the live-range partitioning (step four) are to first limit the instruction imbalance to the range dictated by the value of β, the balance factor (see Section 7.2.1), and then to minimize the number of instructions that are distributed to more than one cluster. The global scheduler, when choosing a cluster for a live range L, takes into account the instruction balance around every instruction that uses live range L. The local scheduler, on the other hand, selects one of the instructions that writes L, and, if necessary, chooses a cluster for L based only on the instruction balance around the chosen instruction. The schedulers are described in detail in the next two sections, beginning with the global scheduler.

A.1.1 Global Scheduler

The global scheduler takes into account the instruction balance at multiple points in the intermediate-language program when choosing a cluster for a live range. The live ranges are visited individually, and based on the instruction distribution in the neighborhood of each instruction that uses the live range, a cluster is chosen for the live range. To precisely quantify instruction balance in the neighborhood of an instruction, it would be necessary to evaluate the impact that every live range has on the balance, and, because instruction balance pertains to the run-time distribution of instructions, it would also be necessary to take into account control flow. But, as discussed in Section 7.2.4.2, owing to the complexity of precisely quantifying instruction balance, it is assumed that the instruction balance in the neighborhood of an instruction depends only on the instructions in the same basic block and the live ranges that these instructions use.

Quantifying Instruction Balance

The instruction balance in the neighborhood of an instruction is represented by a pair of *balance variables* α^a and α^b. The values of these variables for an instruction I are calculated using a top-down, in-order traversal and a bottom-up, in-order traversal of the instructions in the basic block that contains I. The objective of the top-down traversal is to measure the contribution of the instructions that precede I in the basic block, while the objective of the bottom-up traversal is to measure the contribution of the instructions that follow I. Thus, from the perspective of the instruction, the top-down traversal provides information about the past while the bottom-up traversal provides information about the future. The reason for providing both sources of information is discussed below.

If there are n instructions in the basic block, then the value of the α_i^a variable for the i^{th} instruction, $i \leq n$, is equal to the number of instructions in the set $\{1, \ldots, i\}$ that would be distributed only to cluster

C_1, minus the number that would be distributed only to cluster C_2. That is, if N_i^1 is the number of instructions that would be distributed only to cluster C_1 after the i^{th} instruction is visited, and N_i^2 is the number that would be distributed to only cluster C_2 after the i^{th} instruction is visited, then $\alpha_i^a = N_i^1 - N_i^2$. If the absolute value of α_i^a is greater than the balance factor β, that is, $\|\alpha_i^a\| > \beta$, then the instruction distribution is considered unbalanced around the i^{th} instruction.

The α^b balance variables are calculated in a similar fashion but during the bottom-up traversal rather than the top-down traversal. In this case, N_i^1 is the number of instructions in the set $\{i, \ldots, n\}$ that would be distributed to only cluster C_1, while N_i^2 is the number of instructions in the set $\{i, \ldots, n\}$ that would be distributed to only cluster C_2. As before, the instruction distribution around the i^{th} instruction is considered unbalanced if $\|\alpha_i^b\| > \beta$. Note that since the degrees of imbalance are based on running totals, the degree of imbalance of the basic block $\alpha^B = \alpha_1^b = \alpha_n^a$.

Choosing a Cluster Preference for an Instruction

Informally, to choose a cluster for a live range L, the global scheduler determines for each instruction that reads or writes the live range, the best cluster for L. Then, once all the instructions have been visited, the live range is assigned to the cluster that is preferred by the majority. In determining the best cluster for an instruction I, two factors are taken into account: (1) the degree of instruction balance around I, and (2) the cluster assignments for L that will allow I to be distributed to only one cluster.

To illustrate this process, consider the instructions in the third basic block of the control flow graph shown in Figure A.1. Suppose that the live range H has been assigned to cluster C_2, and that the scheduler is visiting instruction 3 to determine the best cluster assignment for live range C. To ensure that instruction 3 is distributed to only one cluster, live range C would have to be assigned to cluster C_2. This assignment, however, is not permitted if the instruction distribution in the neighborhood of instruction 3 is unbalanced. To determine whether the distribution is unbalanced, the values of α_3^a and α_3^b are compared to the balance factor β. If the distribution is unbalanced, the values are compared to each other to determine the cluster that should be chosen for live range C so that the imbalance is reduced. In summary, when the scheduler chooses a cluster for live range C, it first determines if the instruction distribution is balanced, and if not, it then determines how to correct it.

The instruction distribution in the neighborhood of the i^{th} instruction is balanced if the absolute values of the balance variables α^a and α^b are less than or equal to the balance factor β. If each balance variable is plotted in one of the dimensions a two-dimensional plane, then the values for which the distribution is balanced would be enclosed by a box centered at the origin with sides of length 2β.

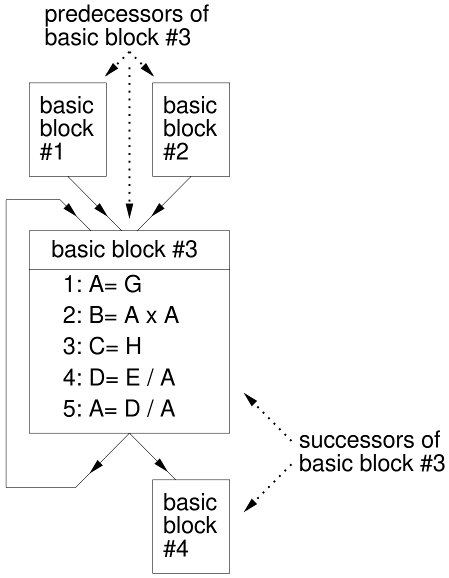


Figure A.1: Control flow graph segment showing in detail basic block #3 and the points at which the instruction balance is calculated.

This box is depicted as a gray rectangle in Figure A.2. In this figure, the α^a variable is plotted on the horizontal axis while the α^b variable is plotted on the vertical axis. The values of these variables for which the distribution is not balanced lie outside of the gray box, and in one of four regions.

Suppose the scheduler is determining the cluster preference of the i^{th} instruction for a live range L. The first region comprises the values for which $\alpha^a > \alpha^b$ and $\alpha^a > \beta$. Since the variable α^a has the bigger value, to reduce the unbalance in the vicinity of the i^{th} instruction, a cluster should be chosen for L so that the value of α^a is decreased. But since $\alpha_i^a = (N_i^1 - N_i^2) > 0$, where N_i^1 and N_i^2 are the number of instructions distributed to only cluster C_1 and C_2 respectively, to reduce the value of α_i^a, the value of N_i^2 must be increased. Thus, L must be assigned to cluster C_2. Similarly, if the point defined by the values of the balance variables lies in the second region, cluster C_2 must again be chosen because here too, increasing the N_i^2 count will decrease the amount of imbalance. On the other hand, if the point defined by the two balance variables lies in the other two regions, by similar reasoning, cluster C_1 must be chosen for live range L.

Note that in making a cluster choice for a live range, the scheduler must take into account both balance variables, since, to use only one could lead to an incorrect cluster choice. Consider, for instance, if $0 < \alpha^a < \beta$ and $\alpha^b > \beta$. In this case, if the scheduler ignored the value of α^b, it might choose cluster C_1 for the live range. But, this choice would increase the value of α^b, and hence, the instruction distribution imbalance. To prevent the scheduler from making such a choice, the instruction distribution balance above the instruction and below the instruction are taken into account.

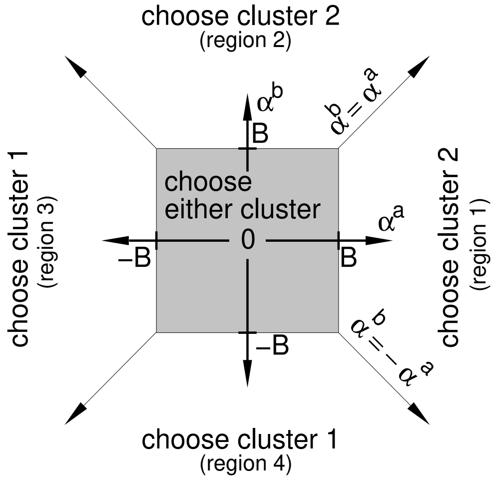


Figure A.2: Graphical representation of the conditions governing which cluster is chosen for a particular set of α values.

Should the distribution be balanced, then the scheduler will chose a cluster C for live range L if the i^{th} instruction will be then distributed to only one cluster. However, if the instruction names two other live ranges, both of which are assigned to different clusters, then some other metric is required to identify the better cluster for L. Two metrics were investigated: (1) the number of master copies of instructions that are distributed to each cluster should be balanced, or (2) the number of instructions that require operand transfer-buffer entries should be balanced. For both metrics, the scheduler used a similar decision process to the one it used for the balance variables. That is, first top-down and bottom-up traversals were carried out to determine the value of the metric before and after each instruction. Then, the conditions depicted in Figure A.2 were used to select the cluster preference of the instruction for L. In addition to these metrics increasing significantly the complexity of the scheduler, in practice it was found that the performance obtained using them was nearly identical to that obtained by not choosing any cluster for instructions that were guaranteed to be dual issued.

Choosing a Cluster for a Live Range

When choosing a cluster for a live range L, the cluster preference of each instruction that uses L is weighed using an estimate of the number of times the instruction is likely to execute. These execution estimates are based on the number of times each (static) instruction commits when the application is run on a dual-cluster processor using the machine code generated by the OSF/1 compiler. This method for estimating the execution frequency of instructions was chosen because the estimates are insensitive to the number of mispredicted branches and the decisions made by the global scheduler. Hence, the performance impact of design tradeoffs could be more easily evaluated.

These execution estimates are also used to quantify the work each live range is likely to do. In this context, work is equal to the number of times a live range is read or written. The live ranges with the largest work estimates are visited first because the performance of an application is more sensitive to the instruction balance of the most frequently executed code segments. Nevertheless, there is no guarantee that the objective of a balanced distribution will be met. The cluster assignment for a live range L depends on the cluster preferences of the instructions that use the live range, but these preferences are sensitive to the order in which the live ranges are visited. The source of this sensitivity is the need to take into account the cluster distribution of instructions in the vicinity of an instruction I when the cluster preference is made by the scheduler for I. Since this distribution can change with the future allocation of any yet unassigned live range, there is no guarantee that the cluster chosen now would be the one chosen in the future.

Summary

In summary, the global scheduler consists of three phases. In the first phase, if some live ranges must be assigned to a specific cluster, say, to ensure that the calling convention for shared libraries is upheld, these live ranges are pre-assigned; pre-assignment is discussed in Section A.2.2. Then, the other live ranges are sorted, and the instruction distribution balance in the neighborhood of each instruction is calculated. In the second phase, the live ranges are assigned to a cluster based on the cluster preference of the instructions that use them. And finally, in the third phase, the live ranges are allocated to the architectural registers. A pseudo-code description of these steps is given in Figure A.3.

A.1.2 Local Scheduler

The global scheduler attempts to correct the instruction imbalance at numerous points in the code, a process that is hindered by the instruction balance at one point being directly and indirectly affected by the balance at another point; this approach can lead to decisions that are neither globally advantageous nor locally advantageous. The local scheduler, on the other hand, sacrifices this attempt at global optimality for more control over instruction balance at specific points.

While the global scheduler seeks an assignment for a live range L that benefits the majority of the instructions that use L, the local scheduler chooses one of these instructions and if necessary, determines the assignment for L without considering any of the other instructions. The instruction that is chosen is referred to as the *pivotal instruction*, or I_p. Whether the other instructions are considered depends on the

```
initialization
    for each live range L
        if live range is to be pre-assigned
            assign it to proper cluster
        else
            determine the work done by L
            add L to list of live ranges which is sorted by the work done
        end if
    end for
    for each basic block B
        compute the balance-variable values for each instruction in B
    end for
cluster assignment
    for each live range L on list
        for each instruction I that reads or writes live range L
            determine the cluster preference of I for L
        end for
        assign L to the most popular cluster
        recalculate the balance variables in the basic blocks in which L is read or written
    end for
register allocation
    for each live range L
        assign L to a register in the desired cluster, or spill it to the other cluster or to memory
    end for
```

Figure A.3: Pseudo-code description of the global scheduler.

instruction balance around I_p. If the instruction balance is within the range defined by the instruction balance factor β, then the other instructions will be considered. But if the balance is not within range, then L will be assigned to the cluster that reduces the degree of imbalance around I_p.

The local scheduler uses the following method to determine the pivotal instruction for live range L. First, it sorts the basic block by assigning a higher rank to those that are estimated to be executed a larger number of times. The execution estimate for a basic block is equal to the number of times the first instruction in the basic block is committed when the application is executed on a dual-cluster processor. The binary used to acquire the execution estimates is the one generated by the OSF/1 compiler. Once all basic blocks are ranked, the scheduler does a bottom-up, in-order traversal of the instructions in each basic block, beginning with the block having the highest rank and ending with the block having the lowest rank. The pivotal instruction for L is the first instruction encountered during the traversals that names L as its destination. That is, instruction I_p is the first instruction encountered that writes L. When this instruction is encountered, L is assigned to a cluster. Once the assignment is made, it cannot be changed. When subsequent writers of L are encountered during the traversal of the basic blocks, these instructions are ignored.

To assign the live range L to a cluster, the scheduler first determines whether the instruction distribution is balanced around the pivotal instruction. To determine whether it is balanced, the values of the α^a and α^b balance variables are compared. This comparison uses the same procedure as is used in the case of the global scheduler. If the instruction distribution is not balanced, the scheduler assigns L to the under-subscribed cluster. But, if the distribution is balanced, the scheduler considers the other instructions that use L before choosing a cluster for L.

Instruction Distribution is Unbalanced Around the Pivotal Instruction

To reduce the instruction distribution imbalance around the pivotal instruction I_p, L should be assigned to cluster C_1 if C_1 is over subscribed, or to cluster C_2 if C_2 is over subscribed. Figure A.2 indicates the conditions under which one cluster or the other is chosen. While fairly straight forward, this strategy of choosing the under-subscribed cluster for L may only result in an increase in the number of dual-distributed instructions while having no effect on the instruction-distribution imbalance. This effect will occur if the pivotal instruction names another live range L_s as a source operand, and L_s has already been assigned to the over-subscribed cluster C_o; this effect will also occur if I_p names two source live ranges, and both have been allocated to C_o. If L is then assigned to the under-subscribed cluster C_u, the instruction will be dual-distributed but the degree of imbalance will not be reduced. Since it is desirable to assign L to cluster C_o so that I_p is not dual-distributed, the following modification to the basic strategy was investigated.

The scheduler is allowed to postpone making a cluster assignment for L until the instruction I_e that precedes I_p in the basic block is visited. Assuming that I_e writes the live range L_e, there are three scenarios to consider:

1. If L_e is unassigned and if this live range can be assigned to the under-subscribed cluster C_u without I_e being dual distributed, then (1) L is assigned to the over-subscribed cluster C_o, and (2) L_e is assigned to the under-subscribed cluster C_u. The net effect of these two assignments is that neither I_p nor I_e is dual distributed.

2. If L_e is assigned and if I_e will be distributed only to the under-subscribed cluster C_u, then L is assigned to the over-subscribed cluster C_o.

3. Otherwise, both L and L_e are assigned to the under-subscribed cluster C_u.

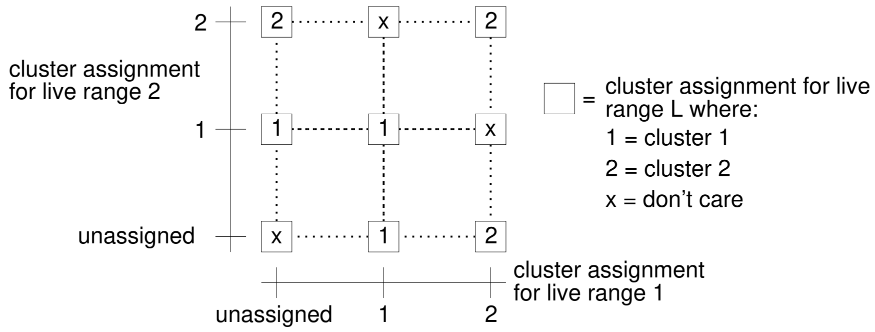


Figure A.4: Cluster assignment for live range L given the cluster assignments for live ranges #1 and #2.

In practice, this modification was found to give slightly better performance but the increase was in general too minor to be significant. While it is possible that a more significant performance gain could have been made if three rather than two instructions were considered, larger numbers of instructions were not considered due to algorithmic complexity.

Instruction Distribution is Balanced Around the Pivotal Instruction

When the instruction distribution is balanced around the pivotal instruction I_p, the cluster for L is chosen by visiting the other instructions that name L as a source or destination. The objective of this process is to determine the cluster assignment for L that will result in the majority of these instructions being distributed to only one cluster. After visiting the instructions, the preferences are tallied, and L is assigned to the cluster that was preferred by the majority.

When an instruction I is visited, the scheduler determines the preferred cluster for I by inspecting the cluster assignments of the one or two other live ranges it names. For convenience, let I name two other live ranges L_1 and L_2. Because L_1 and L_2 may be unassigned, assigned to cluster C_1, or assigned to cluster C_2, there are $3 \times 3 = 9$ cases to consider. The objective of the inspection of these nine cases is to determine the preferred cluster for Lthat would permit instruction I to be distributed to only one cluster. There are three possible outcomes of the inspection: cluster C_1 is preferred, cluster C_2 is preferred, or neither is preferred. The nine cases and the outcomes that apply to each are shown in Figure A.4. In this figure, the horizontal axis indicates the cluster assignment of L_1 while the vertical axis indicates the assignment for L_2. The squares located at the nine points of intersection enclose the outcome chosen for the corresponding (L_1, L_2) pair. For example, if both live ranges are unassigned, or both are assigned but to different clusters, then, there is no preferred cluster for L; an 'x' is used in the figure to represent no preference. As a second example, if L_1 is assigned and L_2 is not, then the preferred cluster is the cluster to which L_1 is assigned; these cases correspond to the row with $L_2 = unassigned$.

Summary

The pseudo code presented in Figure A.5 summarizes the steps taken by the local scheduler. The reader is referred to the discussion of the pseudo code for the global scheduler for an explanation of the pre-assigning of live ranges in the initialization step.

A.2 Implementation Details

This section describes the implementation of the min-cut, global and local schedulers. As discussed in Section 7.3.1, the DEC OSF/1 compilers are used as a front end to generate a *native binary*. This binary is then analyzed using ATOM to discover the data and control dependences between instructions, and the live ranges these instructions read and write.

The analysis uses information extracted from the stream of instructions that are committed when the native binary is executed on a dual-cluster processor. The following information is extracted: (1) the number of times each basic block appears in the stream, (2) the number of times basic block B_1 immediately precedes basic block B_2 in the stream, and (3) the jump targets of all indirect jumps. The jump target information is used to build the control flow graph while the execution counts are used to guide the live range partitioning and allocation process. Jump targets cannot be statically determined because ATOM does not provide access to the symbol table. A symbol table provides a mechanism for determining the targets of jumps that do not implement procedure calls via procedure variables. However, if procedure calls are made via a variable, the symbol table can be used to identify a set of possible targets (Srivastava and Wall use this approach [1992]).

The min-cut, global, and local schedulers are used to partition the live ranges and allocate them to architectural registers. The object code generated by the schedulers is referred to as the *rescheduled binary*. The process of partitioning and allocating is carried out on a per-procedure basis to reduce algorithmic complexity and to avoid problems induced by not having access to the symbol table. That is, each procedure is processed individually. The schedulers rely on ATOM to partition the native binary of an application into procedures. The procedures of are visited in no particular order because, as explained in Section A.2.2, the register calling convention is enforced at procedure boundaries.

A.2.1 Building a Control Flow Graph

For each procedure, a control flow graph [Aho 1986] is built. In this graph, there is a node corresponding to each instruction in the procedure and an edge from an instruction I_1 to an instruction I_2 if I_2 can

```
initialization
    for each basic block B
        add B to sorted list of basic blocks
    end for
    for each live range L
        if live range is to be pre-assigned
            assign it to proper cluster
        end if
    end for
cluster assignment
    for each basic block B_a on list
        for each instruction I_a in B_a from last to first
            if I_a does not write an unassigned live range L
                skip to next instruction
            end if
            if instruction balance is valid
                for each instruction I_b that names live range L
                    nominate a cluster for L
                end for
                assign L to the most popular cluster
            else
                assign L to the under-subscribed cluster
            end if
            for each basic block B_b containing one or more of the instructions that name L
                compute balance variables for each instruction in B_b
            end for
        end for
    end for
register allocation
    for each live range L
        assign L to a register in the desired cluster,or spill it to the other cluster or to memory
    end for
```

Figure A.5: Pseudo-code description of the local scheduler.

immediately follow I_1 in some execution sequence. Instructions are then flagged if they are either (1) the target of a branch from outside of the procedure, that is a point of entry into the procedure, or (2) the source of a branch to another procedure, that is, a point of exit out of the procedure. Entry and exit points are referred to as the *boundaries* of the procedure.

A.2.2 Determining the Live Ranges

Once the control flow graph is built, it is used to solve the live-value and reaching-definition data flow problems [Aho 1986]. The solutions to these problems are then used to determine the live ranges in the procedure. Because inter-procedure data flow information is not available, it is assumed that the OSF/1 calling convention [Digital 1993, Digital 1994] applies at the procedure boundaries. Therefore, the following architectural registers are assumed to contain a valid value on entry to a procedure or on exit from a procedure:

register type	on entry to procedure	on exit from procedure
integer	$R_9, \ldots, R_{21}, R_{26}, R_{27}, R_{29}, R_{30}$	$R_0, R_9, \ldots, R_{15}, R_{26}, R_{29}, R_{30}$
floating point	$F_2, \ldots, F_9, F_{16}, \ldots, F_{21}$	$F_0, F_2, \ldots, F_9$

The calling convention specifies that in a standard procedure call:

- Register R_0 and F_0 are used to return the result of the called procedure;
- Registers $R_9, \ldots, R_{15}, F_2, \ldots, F_9$ must be saved and restored by the called procedure before it can use them;
- Registers $R_{16}, \ldots, R_{21}, F_{16}, \ldots, F_{21}$ may be used to pass the first 12 arguments (6 integer and 6 floating point) to the called procedure;
- Register R_{26} is used to pass and return the return address;
- Register R_{27} may contain the procedure value of the called procedure;
- Register R_{29} contains the global offset table segment pointer which is valid on entry to the procedure but necessarily so on exit from it; and
- Register R_{30} contain the pointer to the top of the current stack.

To ensure the calling standard is maintained at the boundaries of the procedure, live ranges are allocated to their *original registers* if they include any instruction that is either an entry point or an exit point. The original register is the register that the live range occupied in the native binary. Live ranges that include boundary points are said to be *boundary live ranges*.

Thus, unlike in some compilers, a copy operation is not introduced at the entry or exit point of procedures to permit boundary live range to occupy a register that is not live on entry or on exit; the reader is referred to [Briggs 1992] for an example in which such copies are used. The register allocation of boundary live ranges is done by the schedulers in the initialization phase (see the discussion of Figure A.3).

A.2.3 Partitioning the Live Ranges

The live ranges that are neither a boundary live range nor a candidate for a global register are then partitioned. In this step, a mapping between the architectural registers and clusters is required. For the reasons discussed on page 155, it is assumed that even-numbered registers are assigned to cluster C_1 and odd-numbered registers are assigned to cluster C_2. Furthermore, since applications tend to read the stack and global pointers much more frequently than they are written, the two registers used to hold these values are assumed to be global. The zero registers[1], R_{31} and F_{31} are also designated global since there is no hardware cost of doing so. All other architectural registers are assumed to be local. As a result of these two assumptions, the architectural registers are assigned as follows:

	cluster C_1	cluster C_2
local	R_0,R_2,$R_4, \ldots,$ R_{28}, F_0,F_2,$F_4, \ldots,$ F_{30}	R_1,R_3,$R_5, \ldots,$ R_{27}, F_1,F_3,$F_5, \ldots,$ F_{29}
global	R_{29}, R_{30}, R_{31}, F_{31}	

Because there is a one-to-one correspondence between the live ranges associated with the global and stack pointers and the global registers R_{29} and R_{30}, the live ranges are allocated to the global registers by the schedulers in the initialization phase. There is a one-to-one correspondence because the live ranges associated with the stack pointer and global pointer are derived from object code to which register allocation has already been applied.

A.2.4 Allocating Live Ranges to Architectural Registers and Generating the Rescheduled Binary

Once the live ranges that are candidates of local registers have been partitioned, they are allocated to the architectural registers. As discussed in Section 7.2.3.2, register allocation is carried out using a modified

[1] A zero register is a special architectural register with the following properties: when it is read, a value of zero is always returned, and when it is written, the value being written is discarded.

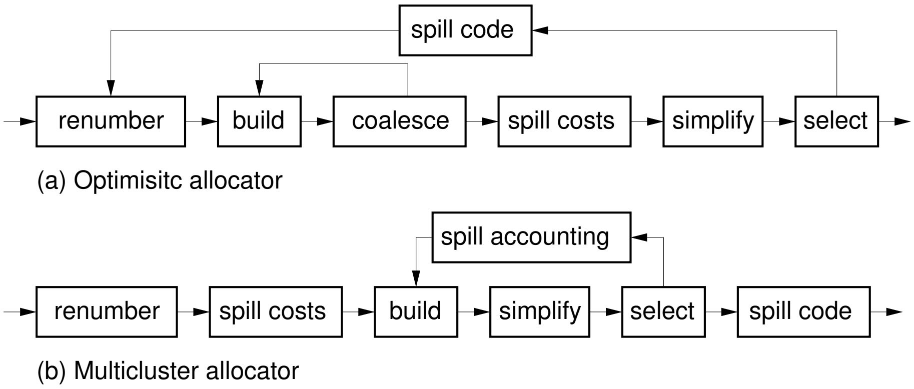


Figure A.6: The register allocator proposed by Briggs et al. and the adapted version for the min-cut, global, and local schedulers.

version of the graph-coloring algorithm developed for the ParaScope Project [Briggs 1992].

The basis of graph-coloring algorithms is an *interference graph*. In this graph, the nodes represent live ranges and the edges represent interference. There is an edge between a node n_1 and a node n_2 if the live range associated with n_1 interferes with the live range associated with n_2. That is, both live ranges are simultaneously live and thus cannot occupy the same architectural register. The live ranges that interfere with a live range L are called the *neighbors* of L. The number of neighbors of L is called its *degree*.

The problem solved by graph coloring is to color the nodes of the interference graph with k colors so that neighboring nodes have different colors. By choosing k to match the number of architectural registers, a k-coloring will represent a feasible allocation of live ranges to registers. Heuristics are used to rank the nodes so that the higher priority ones are colored first, and to select a live range to be spilled should there be no color available for a node. Should spilling be necessary, the graph is modified, and another attempt is made at coloring it. This process repeats until a k-coloring is obtained.

The *optimistic allocator* developed by Briggs et al. is illustrated in Figure A.6(a). It proceeds in seven stages.

1. In *renumber*, the allocator uses the solution to the reaching-definition and live-value problems to determine the set of live ranges.

2. In *build*, the allocator constructs the interference graph.

3. In *coalesce*, the allocator attempts to shrink the number of live ranges. Two live ranges L_1 and L_2 are combined if the initial definition of L_2 is a copy from L_1 and the live ranges do not otherwise interfere. Combining these two live ranges eliminates a copy instruction.

4. In *spill costs*, the allocator estimates, for each live range, the run-time costs of any instructions that would have to be added if the live range were spilled to memory.

5. In *simplify*, the allocator removes from the interference graph nodes that have degree less than k in arbitrary order. It then adds them to a FIFO queue, which is referred to as the *to-allocate queue*. Whenever the allocator discovers that all the remaining nodes in the graph have a degree greater than k, it chooses a live range to spill. The node for the chosen live range is removed from the graph, and is pushed on to the queue. The hope is that a color will be available for the live range when colors are assigned in the next stage.

6. In *select*, the allocator removes the live range at the head of the to-allocate queue, inserts its corresponding node into the interference graph, and assigns the node a color that is distinct from the already-placed neighbors. This process is repeated until the to-allocate queue is empty. If all nodes receive colors, the allocation has succeeded. If any nodes are uncolored, the allocator inserts spill code, rebuilds the interference graph, and tries again.

7. In *spill code*, a live range that has to be spilled is converted into a collection of tiny live ranges by inserting loads before uses and stores after definitions, as required.

The underlying premise of the optimistic allocator is the following: a live range with fewer than k neighbors will not have to be spilled, but one with more than k neighbors may have to be spilled depending on how the k colors are assigned to its neighbors. It is for this reason that the optimistic allocator separates the simplify stage from the select stage.

A number of modifications were made to the optimistic allocator to take into account features of the multicluster architecture and to simplify the resulting *multicluster allocator*. The multicluster allocator is illustrated in Figure A.6(b). The modifications are described below.

- The *coalesce* stage is not implemented by the multicluster allocator to simplify it. To support coalescing, the allocator must be capable of inserting new instructions into the procedure. The purpose of these instructions is to move a live range to and from the register(s) that it must occupy across the boundaries of the procedure. When such instructions are added, the live ranges must be recreated for each iteration, and possibly repartitioned.

- In *simplify*, the multicluster allocator removes from the interference graph, nodes for which $N_{partition} > R_{partition}$. For any live range L that has been assigned to partition C by the live-range partitioner (see Section A.2.3):

– N_C is the number of neighbors of live range L that are still in the graph and that have also assigned to partition C. N_C is referred to as the *per-cluster degree*.

– R_C is the number of local registers that are assigned to cluster C. For example, if L is assigned to cluster C_2 and it represents an integer value, based on the data in the table in Section A.2.3, $R_{C_2} = 14$. But if L represents a floating-point value, $R_{C_2} = 15$.

Whenever the allocator discovers that all the remaining nodes have a per-cluster degree greater than $R_{partition}$, it chooses a live range to spill. In choosing a live range to spill, the allocator compares the product of the per-cluster degree and the spill cost (see below). This comparison metric is suggested by Briggs et al. [1994] owing to the frequency that spill selections are made and the number of comparisons that must be carried out for each selection. The node with the largest product is removed from the graph.

- In *select*, the allocator removes the live range at the head of the to-allocate queue. Then, it attempts to allocate this live range to a register that has not been allocated to any of its neighbors. If the live range, call it L, has been assigned to cluster C_1, the allocator tries to allocate L to one of the registers that is assigned to cluster C_1. If there is an available register, the allocator inserts the node corresponding to L into the interference graph, and continues with the next live range from the to-allocate queue. On the other hand, if all the registers in C_1 have already been allocated to the neighbors of L, the register allocation for L is deferred by placing it in a second FIFO queue. This second queue is referred to as the *spill* queue. This queue indicates the live ranges that will be spilled either to a register in the other cluster, or to a memory location.

  When the to-allocate queue is empty, the multicluster allocator attempts to allocate each live range on the spill queue to a register. For a live range L that has been assigned to cluster C_1, the allocator first tries to allocate L to an available register from cluster C_2. If such a register exists, the allocator inserts the node corresponding to L into the interference graph, and continues with the next live range in the spill queue.

  If no such register exists, the allocator tries to change the register allocation of one of the neighbors of L so as to free up a register for L. The strategy the multicluster allocator uses for reallocating registers is based on the one proposed by Briggs et al. in [1994]. When no register exists, allocator determines the set of neighbor live ranges $\mathcal{L}$ for live range L, where $\mathcal{L}$ is the neighbors of L that have been allocated to a register that has not been allocated to any of the other neighbors of L. That is, each neighbor in the set has been allocated to a unique register. If this set is not empty, the allocator tries to reallocate one of the live ranges in the set to a register that is used by another neighbor. However, a neighbor n that has been assigned to register R_a can only be reallocated to another register R_b if R_b belongs to the cluster to which n was assigned by the partitioner. If more than one neighbor can be reallocated, preference is given to the one that enables L to be assigned to the cluster chosen for it by the partitioner. If a neighbor is reallocated, L is then allocated to R_a.

Finally, if reallocation fails, the allocator marks the live range for spilling to memory, and the allocator continues with the next live range in the spill queue. Once the spill queue is empty, if any live ranges were marked for spilling, the allocator modifies these live ranges (see the next stage), rebuilds the interference graph, and tries again. Note, a live range that is spilled to memory still requires a register, but it requires the register for much shorter intervals, and hence, spilling a live range reduces the interference between live ranges. If all nodes receive colors, the allocation has succeeded, and the allocator enters the *spill code* stage.

- In *select*, when choosing a register for a live range L, the allocator attempts to minimize the number of registers that are live in the procedure, and to maximize the likelihood that a live range will be allocated a register in the desired cluster. To meet these objectives, the allocator compiles six sets. The first two, ADJ_{avail}, and ADJ_{avoid} are compiled by visiting the neighbor live ranges of L. The second two, SRC_{used} and SRC_{avail}, are compiled by visiting the instructions that name L as a destination; these two sets are used in an attempt to mimic coalescing. The fifth set, $PROC_{used}$ is compiled by visiting the already-allocated live ranges in the procedure. Finally, the sixth set, R_{avail}, is compiled by also visiting the neighbor live ranges of L.

R_{avail} comprises the registers assigned to the desired cluster for L and to which the neighbors of L have not been allocated.

ADJ_{avail} comprises the registers to which the unallocated neighbors of L could be allocated.

ADJ_{avoid} comprises the registers in the set ADJ_{avail} that represent the only allocation option for a neighbor of L. The allocator tries to avoid allocating L to one of the registers in this set because in so doing, one of the neighbors of L could then not be allocated to a register.

SRC_{used} comprises the registers that have been allocated to the live ranges that do not interfere with L and that are named as source operands by the instructions that write L. The allocator compiles this set by visiting each instruction I that writes L, and recording the register allocated to a live range L_s if L_s satisfies the following: (1) it is named by I as a source operand, (2) it does not interfere with L, (3) it has been assigned to the same cluster as L, and (4) it requires the same type of register as L.

SRC_{avail} comprises the registers to which the unallocated source-operand live ranges could be allocated. The allocator compiles this set by visiting each instruction I that writes L, and recording the registers that could be used for an unallocated source live range L_s if L_s satisfies the following: (1) it is named by I as a source operand, (2) it does not interfere with L, (3) it has been assigned to the same cluster as L, and (4) it requires the same type of register as L.

$PROC_{used}$ comprises the registers to which live ranges in the procedure have already been allocated.

The allocator determines the set of possible registers for L, $R_{possible}$, from the set of those that could be used for L, R_{avail}, in two phases. In the first phase, the allocator applies in succession the selection strategies listed below until a non-empty set $R_{possible}$ is obtained. In this phase, it uses those registers to which L could be allocated and that have already been allocated to another live range in the procedure. In this way, the allocator attempts to minimize the number of live registers in the procedure. Thus, $R_{pool} = R_{avail} \cap PROC_{used}$. If no register is selected, the allocator proceeds with the second phase. In this phase, the allocator does not include the stipulation that the registers must already be allocated to a live range in the procedure. Thus, $R_{pool} = R_{avail}$.

1	$R_{possible} = R_{pool} \cap \overline{ADJ_{avail}}$	• registers not available to the neighbors of L
2	$R_{possible} = R_{pool} \cap SRC_{used} \cap \overline{ADJ_{avoid}}$	• registers allocated to non-interfering live ranges and not in the avoid set
3	$R_{possible} = R_{pool} \cap SRC_{avail} \cap \overline{ADJ_{avoid}}$	• registers to which non-interfering live ranges could be allocated and not in the avoid set
4	$R_{possible} = R_{pool} \cap \overline{ADJ_{avoid}}$	• registers available to L but not in the avoid set
5	$R_{possible} = R_{pool} \cap SRC_{used}$	• registers allocated to non-interfering live ranges
6	$R_{possible} = R_{pool} \cap SRC_{avail}$	• registers to which non-interfering live ranges could be allocated
7	$R_{possible} = R_{pool}$	• registers available to L

The allocator uses random selection to chose a register for L from the set $R_{possible}$.

- Unlike in the case of the optimistic allocator, a live range that is to be spilled is not converted into a collection of tiny live ranges. Rather, the multicluster allocator changes the definition of the live range so that the live range is live only immediately before a use and immediately after a definition; the *spill accounting* stage performs this task.

The live range is changed rather than converted to avoid having to recreate the live ranges, and possibly repartition them. Because the live range is not converted into a collection of tiny live ranges, if the multicluster allocator wants to spill a live range, it must spill all of it[2]. On the other hand, the optimistic allocator may spill only part of a live range to memory.

- Because the live ranges were not converted into a collection of tiny ranges, the spill cost for each live range is computed only once. Thus, the *spill cost* stage is located outside of the graph-coloring loop, as shown in Figure A.6(b). The spill cost of a live range is estimated as the sum of the estimated execution counts of all the instructions that either read or write the live range.

- The *spill code* stage is performed at the same time that the native object code is instrumented with ATOM and linked with the multicluster simulator. As the instructions are instrumented, if an instruction reads a live range that has been marked for spilling to memory, a load instruction is inserted before the instruction. Likewise, a store instruction is added after an instruction if it writes a live range that has been marked for spilling to memory.

  Since these spill instructions did not exist in the native binary, the following two assumptions are made:

  1. The program-counter address of the required load and store instructions for implementing the spill is assumed to be the program-counter address of the instruction that reads or writes the live range being spilled.

  2. The address to which a live range is spilled is determined at run time and is not deterministic. The address from which the live range is read or to which it is written is the last stack address accessed by any load and store instruction that is not implementing a live-range spill plus a random offset between 0 and 1023. Thus, each time the live range is read from memory or written to memory, a different address may be used.

[2] A second strategy was also tried in which some of the loads and stores are avoided by keeping the live range live between two uses or a use and a definition. Chow and Hennessy [1990] describe a set of conditions which can be used to determine whether the live range can exist in an interval. In practice, this more complex strategy was found to have little performance impact because the three schedulers did not spill very many live ranges.

Bibliography

Abraham, S. G., Sugumar, R. A., Windheiser, D., Rau, B. R., and Gupta, R. (1993). Predictability of load/store instruction latencies. *In the Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 139–152.

Agrawal, A. (1994). UltraSPARC: A 64-Bit, High-Performance SPARC Processor. *In the Proceedings of MicroProcessor Forum*.

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers, Principles, Techniques and Tools*. Addison-Wesley Publishing Company, Reading Mass.

Allan, V. H., Jones, R. B., Lee, R. M., and Allan, S. J. (1995). Software Pipelining. *ACM Computing Surveys*, pages 367–432.

Anderson, D. W., Sparacio, F. J., and Tomasulo, R. M. (1967). The IBM 360 Model 91: Machine Philosophy and Instruction Handling. *IBM Journal of Research and Development*, 11(1):8–24.

Bacon, D. F., Graham, S. L., and Sharp, O. J. (1993). Compiler Transformations for High-Performance Computing. Technical Report UCS/CSD-93-781, Computer Science Division, University of California at Berkeley.

Bradlee, D., Eggers, S., and Henry, R. (1991). The Effect on RISC Performance of Register Set Size and Structure Versus Code Generation Strategy. *In the Proceedings of the 18th International Symposium on Computer Architecture*, pages 330–339.

Briggs, P. (1992). *Register Allocation via Graph Coloring*. PhD thesis, Department of Computer Science, Rice University, Houston, Texas.

Briggs, P., Cooper, K. D., and Torczon, L. (1994). Improvements to Graph Coloring Register Allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455.

Butler, M. and Patt, Y. (1992). An Investigation of the Performance of Various Dynamic Scheduling Techniques. *In the Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 1–9.

Calder, B. and Grunwald, D. (1994). Reducing Branch Costs via Branch Alignment. *In the Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 242–251.

Callahan, D., Kennedy, K., and Porterfield, A. (1991). Software Prefetching. *In the Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52.

Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., and Markstein, P. W. (1981). Register allocation via coloring. *Computer Languages*, 6(1):47–57.

Chang, P. P., Mahlke, S. A., Chen, W. Y., Warter, N. J., and Hwu, W. W. (1991). IMPACT: an Architectural Framework for Multiple-Instruction-Issue Processors. *In the Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 266–275.

Chen, T.-F. and Baer, J.-L. (1992). Reducing Memory Latency via Non-blocking and Prefetching Caches. *In the Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61.

Chow, F. and Hennessy, J. (1990). The Priority-Based Coloring Approach to Register Allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536.

Christie, D. (1996). Developing the AMD-K5 Architecture. *IEEE Micro*, 16(2):16–26.

Digital (1993). *DEC OSF/1 Assembly Language Programmer's Guide*. Digital Equipment Corporation.

Digital (1994). *DEC OSF/1 Calling Standard for AXP Systems*. Digital Equipment Corporation.

Dwyer, H. and Torng, H. C. (1992). An Out-of-Order Superscalar Processor with Speculative Execution and Fast, Precise Interrupts. *In the Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 272–281.

Edmondson, J. H., Rubinfeld, P. I., Bannon, P. J., Benschneider, B. J., Bernstein, D., Castelino, R. W., Cooper, E. M., Dever, D. E., Fischer, D. R. D. T. C., Jain, A. K., Mehta, S., Meyer, J. E., Preston, R. P., Rajagopalan, V., Somanathan, C., Taylor, S. A., and Wolrich, G. M. (1995). Internal Organization of the Alpha 21164, a 300-MHZ 64-bit Quad-issue CMOS RISC Microprocessor. *Digital Technical Journal*, 7(1):119–132.

Farkas, K. I. and Jouppi, N. P. (1994). Complexity/Performance Tradeoffs with Non-Blocking Loads. *In the Proceedings of the 21st International Symposium on Computer Architecture*, pages 211–222.

Farkas, K. I., Jouppi, N. P., and Chow, P. (1995). How Useful Are Non-blocking Loads, Stream Buffers and Speculative Execution in Multiple Issue Processors? *In the Proceedings of the First International Symposium on High Performance Computer Architecture*, pages 78–89.

Franklin, M. and Sohi, G. (1992). Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors. *In the Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 236–245.

Fu, J. W., Patel, J. H., and Janssens, B. L. (1992). Stride Directed Prefetching in Scalar Processors. *In the Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 102–110.

Gallagher, D. (1995). *Memory Disambiguation to Facilitate Instruction-Level Parallelism Compilation*. PhD thesis, Department of Computer Science, University of Illinois, Urbana IL.

Gibbons, A. M. (1985). *Algorithmic Graph Theory*. Cambridge University Press, New York.

Gwennap, L. (1994a). MIPS R10000 Uses Decoupled Architecture. *Microprocessor Reports*, 8(14):18–22.

Gwennap, L. (1994b). PA-8000 Combines Complexity and Speed. *Microprocessor Reports*, 8(15):1,6–11.

Gwennap, L. (1995). Intel's P6 Uses Decoupled Superscalar Design. *Microprocessor Reports*, 9(2):9–15.

Gwennap, L. (1996). Digital 21264 Sets New Standard. *Microprocessor Report*, 10(14).

Halfhill, T. R. (1996). AMD K6 Takes On Intel P6. *BYTE*, (1):67–72.

Hendren, L. J., Gao, G. R., Altman, E. R., and Mukerji, C. (1993). A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs. *The Journal of Programming Languages*, 1(3):155–185.

IBM (1994). *PowerPC 620 RISC Microprocessor Technical Summary*. IBM Microelectronics. document number: MPR620TSU-01.

Johnson, M. (1991). *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, New Jersey.

Jouppi. Private communication with Norman Jouppi, Digital Equipment Corporation Western Research Lab.

Jouppi, N. (1990). Improving Direct Mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers. Technical Report TN-15, Digital Equipment Corporation Western Research Lab.

Jouppi, N. P. (1993). Cache Write Policies and Performance. *In the Proceedings of the 20th International Symposium on Computer Architecture*, pages 191–201.

Kerns, D. R. and Eggers, S. J. (1993). Balanced Scheduling: Instruction Scheduling When Memory Latency is Uncertain. *In the Proceedings of the Conference on Programming Language Design and Implementation*, pages 278–289.

Kroft, D. (1981). Lockup-Free Instruction Fetch/Prefetch Cache Organization. *In the Proceedings of the Eighth International Symposium on Computer Architecture*, pages 81–87.

Lam, M. S. (1988). Software Pipelining: An Effective Scheduling Technique for VLIW Machines. *In the proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328.

Lengauer, T. (1990). *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, Chichester.

Liptay, J. S. (1992). Design of the IBM Enterprise System/9000 High-end Processor. *IBM Journal of Research and Development*, 36(4):713–731.

Lowney, P. G., Freudenberger, S., Karzes, T., Lichtenstein, W., Nix, R. P., O'Donnell, J. S., and Ruttenberg, J. C. (1993). The Multiflow Trace Scheduling Compiler. *Journal Of Supercomputing*, 7(1-2):51–142.

Luk, C.-K. and Mowry, T. C. (1996). Compiler-Based Prefetching for Recursive Data Structures. *In the Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233.

McFarling, S. (1993). Combining Branch Predictors. *Digital Equipment Corporation Western Research Lab Technical Note TN-36*.

Mowry, T. C. (1994). *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Department of Electrical Engineering, Stanford University, Stanford CA.

Mowry, T. C., Lam, M. S., and Gupta, A. (1992). Design and Evaluation of a Compiler Algorithm for Prefetching. *In the Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73.

Palacharla, S. and Kessler, R. E. (1994). Evaluating Stream Buffers as a Secondary Cache Replacement. *In the Proceedings of the 21st International Symposium on Computer Architecture*, pages 24–33.

Patterson, D. and Hennessy, J. (1995). *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers, 2nd edition.

Proebsting, T. A. and Fischer, C. N. (1992). Probabilistic Register Allocation. *In the Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 300–310.

Rau, B. R. and Fisher, J. A. (1993). Instruction-Level Parallel Processing: History, Overview, and Perspective. *Journal of Supercomputing*, 7(1):9–50.

Sites, R. L. (1992). *Alpha Architecture Reference Manual*. Digital Press.

Sites, R. L. (1993). Alpha AXP Architecture. *Communications of the ACM*, 36(2):33–44.

Smith, J. E. and Pleszkun, A. R. (1985). Implementation of Precise Interrupts in Pipeline Processors. *In the Proceedings of the 12th International Symposium on Computer Architecture*, pages 36–44.

Smotherman, M., Krishnamurthy, S., Aravid, P., and Hunnicutt, D. (1991). Efficient DAG Construction and Heuristic Calculation for Instruction Scheduling. *In the Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 93–102.

Sohi, G. S., Breach, S. E., and T. N. Vijaykumar (1995). Multiscalar Processors. *In the Proceedings of the 22st International Symposium on Computer Architecture*, pages 414–425.

Song, S. P., Denman, M., and Chang, J. (1994). The PowerPC 604 RISC Microprocessor. *IEEE Micro*, 14(5):8–17.

Srivastava, A. and Eustace, A. (1994). Atom: A system for building customized program analysis tools. *In the Proceedings of the ACM SIGPLAN '94 Conference on Programming Languages*.

Srivastava, A. and Wall, D. W. (1992). A Practical System for Intermodule Code Optimization at Link-Time. Technical Report 92/6, Digital Equipment Corporation Western Research Lab.

Tullsen, D. M., Eggers, S. J., Emer, J. S., Levy, H. M., Lo, J. L., and Stamm, R. L. (1996). Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreaded Processor. *In the Proceedings of the 23rd International Symposium on Computer Architecture*, pages 191–202.

Wall, D. W. (1993). Limits of Instruction-Level Parallelism. Technical Report 93/6, Digital Equipment Corporation Western Research Lab.

Wilton, S. J. E. and Jouppi, N. P. (1994). An Enhanced Access and Cycle Time Model for On-Chip Caches. Technical Report 93/5, Digital Equipment Corporation Western Research Lab.

Yeager, K. C. (1996). The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40.