Super-Scalar Processor Design

William M. Johnson

Technical Report No. CSL-TR-89-383

June 1989

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 943054055

Abstract

A super-scalar processor is one that is capable of sustaining an instruction-execution rate of more than one instruction per clock cycle. Maintaining this execution rate is primarily a problem of scheduling processor resources (such as functional units) for high utilization. A number of scheduling algorithms have been published, with wide-ranging claims of performance over the single-instruction issue of a scalar processor. However, a number of these claims are based on idealizations or on special-purpose applications.

This study uses trace-driven simulation to evaluate many different super-scalar hardware organizations. Super-scalar performance is limited primarily by instruction-fetch inefficiencies caused by both branch delays and instruction misalignment. Because of this instruction-fetch lirnitation, it is not worthwhile to explore highly-concurrent execution hardware, Rather, it is more appropriate to explore economical execution hardware that more closely matches the instruction throughput provided by the instruction fetcher. This study examines techniques for reducing the instruction-fetch inefficiencies and explores the resulting hardware organizations.

This study concludes that a super-scalar processor can have nearly twice the pexformance of a scalar processor, but that this requires that four major hardware features: out-of-order execution, register renarmng, branch prediction, and a four-instruction decoder. These features are interdependent, and removing any single feature reduces average performance by 18% or more. However, there are many hardware simplifications that cause only a small performance reduction.

Key Words and Phrases: super-scalar, instruction-level concurrency, instruction-level parallelism, fine-grained parallelism, instruction scheduling, hardware scheduling, register renaming, branch prediction.

Petitioner Lenovo (United States) Inc. - Ex. 1012

Copyright 01989

by

William M. Johnson

Table of Contents

Chapter 1 Introduction 1

Chapter 2 Background 5
- 2.1 Fundamental Concepts 5
  - 2.1.1 Instruction Scheduling Policies 5
  - 2.1.2 Scheduling Constraints 8
  - 2.1.3 Storage Conflicts and Register Renaming 10
- 2.2 Published Techniques for Single-Instruction Issue 11
  - 2.2.1 Common Data Bus-Tomasulo's Algorithm 12
  - 2.2.2 Derivatives of Tomasulo's Algorithm 13
- 2.3 Published Techniques for Multiple-Insauction Issue 15
  - 2.3.1 Detecting Independent Instructions with a Pm-Decode Stack 15
  - 2.3.2 Ordering Matrices 16
  - 2.3.3 Concurrency Detection in Directly-Executed Languages 17
  - 2.3.4 Dispatch Stack 18
  - 2.3.5 High Performance Substrate 19
  - 2.3.6 Multiple-Instruction Issue with the CRAY–1 Architecture 20
- 2.4 Observations and Conclusions 20

Chapter 3 Methodology and Potential Performance 22
- 3.1 Simulation Technique 22
- 3.2 Benchmark Programs 24
- 3.3 Initial Processor Model 26
  - 3.3.1 Basic Organization 26
  - 3.3.2 Implementation Description 28
  - 3.3.3 Processor Configuration 31
- 3.4 Results Using an Ideal Instruction Fetcher 33

Chapter 4 Instruction Fetching and Decoding 35
- 4.1 Branches and Instruction-Fetch Inefficiencies 35
- 4.2 Improving Fetch Efficiency 38
  - 4.2.1 Scheduling Delayed Branches 38
  - 4.2.2 Branch Prediction 40
  - 4.2.3 Aligning and Merging 41
  - 4.2.4 Simulation Results and Observations 43
- 4.3 Implementing Hardware Branch Prediction 46
  - 4.3.1 Basic Organization 47
  - 4.3.2 Setting and Interpreting Cache Entries 48
  - 4.3.3 Predicting Branches 49
  - 4.3.4 Hardware and Performance Costs 50
- 4.4 Implementing a Four-Instruction Decoder 52

4.4.1 Implementing a Hardware Register-Port Arbiter 54
4.4.2 Limiting Register Access Via Instruction Format 56
4.5 Implementing Branches 58
4.5.1 Number of Pending Branches 59
4.5.2 Order of Branch Execution 59
4.5.3 Simplifying Branch Decoding 60
4.6 Observations and Conclusions 62

Chapter 5 Operand Management 64
5.1 Buffering State Information for Restart 64
5.1.1 Sequential, Look-Ahead, and Architectural State 65
5.1.2 Checkpoint Repair 66
5.1.3 History Buffer 67
5.1.4 Reorder Buffer 68
5.1.5 Future File 69
5.2 Restart Implementation and Effects on Performance 71
5.2.1 Mispredicted Branches 71
5.2.2 Exceptions 74
5.2.3 Effect of Restart Hardware on Performance 75
5.3 Dependency Mechanisms 77
5.3.1 Value of Register Renaming 77
5.3.2 Register Renaming with a Reorder Buffer 78
5.3.3 Renaming with a Future File: Tomasulo's Algorithm 79
5.3.4 Other Mechanisms to Resolve Anti-Dependencies 79
5.3.5 Other Mechanisms to Resolve Output Dependencies 80
5.3.6 Partial Renaming 82
5.4 Result Buses and Arbitration 84
5.5 Result Forwarding 85
5.6 Implementing Renaming with a Reorder Buffer 88
5.6.1 Allocating Processor Resources 88
5.6.2 Instruction Decode 90
5.6.3 Instruction Completion 92
5.7 Observations and Conclusions 93

Chapter 6 Instruction Scheduling and Issuing 95
6.1 Reservation Stations 97
6.1.1 Reservation Station Operation 97
6.1.2 Performance Effects of Reservation-Station Size 98
6.2 Central Instruction Window 100
6.2.1 The Dispatch Stack 101
6.2.2 The Register Update Unit 103
6.2.3 Using the Reorder Buffer to Simplify the Central Window 105
6.2.4 Operand Buses 108
6.2.5 Central Window Implementation 110

6.3 Loads and Stores 111
6.3.1 Total Ordering of Loads and Stores 113
6.3.2 Load Bypassing of Stores 113
6.3.3 Load Bypassing with Forwarding 115
6.3.4 Simulation Results 115
6.3.5 Implementing Loads and Stores with a Central Instruction Window . 116
6.3.6 Effects of Store Buffer Size 119
6.3.7 Memory Dependency Checking 119
6.4 Observations and Conclusions 121

Chapter 7 Conclusion 123
7.1 Major Hardware Features 123
7.2 Hardware Simplifications 125
7.3 Future Directions 127

References 129

List of Tables

Table 1. Benchmark Program Descriptions 25
Table 2. Comparisons of Scalar and Super-Scalar Pipelines 31
Table 3. Configuration of Functional Units 32
Table 4. Estimate of Register-Port Arbiter Size: Two Logic Levels 55
Table 5. Critical Path for Central-Window Scheduling 111
Table 6. Performance Advantage of Major Processor Features 124
Table 7. Cumulative Effects of Hardware Simplifications 125

List of Figures

Figure 1. Simple Definition of Super-Scalar Processor 2

Figure 1. Simple Definition of Super-Scalar Processor 2

Figure 2. Super-Scalar Pipeline with In-Order Issue and Completion 6

Figure 3. Super-Scalar Pipeline with In-Order Issue and Out-of-Order Completion 6

Figure 4. Super-Scalar Pipeline with Out-of-Order Issue and Completion 8

Figure 5. Flowchart for Trace-Driven Simulation 23

Figure 6. Sample Instruction-Issue Distribution 27

Figure 7. Block Diagram of Processor Model 29

Figure 8. Potential Speedup of Three Scheduling Policies, using Ideal Instruction Fetcher 33

Figure 9. Speedups with Ideal Instruction Fetcher and with Instruction Fetcher Modeled after Scalar Fetcher 34

Figure 10. Sequence of Two Instruction Runs for Illustrating Decoder Behavior . . 36

Figure 11. Sequence of Instructions in Figure 10 Through Two-Instruction and Four-Instruction Decoders 36

Figure 12. Dynamic Run Length Distribution for Taken Branches 37

Figure 13. Branch Delay and Penalty Versus Speedup 39

Figure 14. Sequence of Instructions in Figure 11 Through Two-Instruction and Four-Instruction Decoders with Branch Prediction 40

Figure 15. Sequence of Instructions in Figure 11 with Single-Cycle Delay for Decoding Branch Prediction 41

Figure 16. Fetch Efficiencies for Various Run Lengths 42

Figure 17. Sequence of Instructions in Figure 10 Through Two-Instruction and Four-Instruction Decoders with Branch Prediction and Aligning 43

Figure 18. Sequence of Instructions in Figure 10 Through Two-Instruction and Four-Instruction Decoders with Branch Prediction, Aligning, and Merging 43

Figure 19. Speedups of Fetch Alternatives with Two-Instruction Decoder 44

Figure 20. Speedups of Fetch Alternatives with Four-Instruction Decoder 44

Figure 21. Average Branch Target Buffer Hit Rates 46

Figure 22. Instruction Cache Entry for Branch Prediction 48

Figure 23. Example Cache Entries for Code Sequence of Figure 14 49

Figure 24. Performance Decrease Caused by Storing All Branch Predictions with Cache Blocks 51

Figure 25. Performance Degradation with Single-Port Cache Tags 52

Figure 26. Register Usage Distribution of a Four-Instruction Decoder—No Branch Delays 53
Figure 27. Performance Degradation Caused by Limiting a Four-Instruction Decoder to Four Register-File Ports 54
Figure 28. Format for Four-Instruction Group Limiting the Number of Registers Accessed 56
Figure 29. Example Operand Encoding using Instruction Format of Figure 28 57
Figure 30. One Approach to Handling a Branch Target Within Instruction Group 58
Figure 31. Reducing the Number of Outstanding Branches 60
Figure 32. Performance Increase by Executing Multiple Correctly-Predicted Branches Per Cycle 61
Figure 33. Performance Decrease Caused by Computing One Branch Target Address Per Decode Cycle 62
Figure 34. Illustration of Sequential, Look-Ahead, and Architectural State 65
Figure 35. History Buffer Organization 68
Figure 36. Reorder Buffer Organization 69
Figure 37. Future File Organization 70
Figure 38. Correcting State After a Mispredicted Branch 72
Figure 39. Distribution of Additional Branch Delay Caused by Waiting for a Mispredicted Branch to Reach the Head of the Reorder Buffer 73
Figure 40. Performance Degradation Caused by Waiting for a Mispredicted Branch to Reach the Head of the Reorder Buffer 74
Figure 41. Effect of Reorder-Buffer Size on Performance: Allocating for Every Instruction 76
Figure 42. Reducing Concurrency by Eliminating Register Renaming 78
Figure 43. Performance Advantage of Eliminating Decoder Stalls for Output Dependencies 81
Figure 44. Performance Advantage of Partial Renaming 83
Figure 45. Integer Result-Bus Utilization at High Performance Levels 84
Figure 46. Effect of the Number of Result Buses on Performance 85
Figure 47. Distribution of Number of Operands Supplied by Forwarding, Per Result 86
Figure 48. Performance of Direct Tag Search for Various Numbers of List Entries 87
Figure 49. Allocation of Result Tags, Reorder-Buffer Entries, and Reservation-Station Entries 89
Figure 50. Instruction Decode Stage 91
Figure 51. Location of Central Window in Processor (Integer Unit) 96
Figure 52. Performance Effect of Reservation-Station Size 98

Figure 53. Effect of Reservation Station Size on the Average Instruction-Fetch Bandwidth Lost Because of a Full Reservation Station, Per Integer Functional Unit 99
Figure 54. Compressing the Dispatch Stack 102
Figure 55. Performance Effect of Dispatch-Stack Size 102
Figure 56. Register Update Unit Managed as a FIFO 104
Figure 57. Performance Degradation of Register Update Unit Compared to Dispatch Stack 104
Figure 58. Distribution of Decode-to-Issue Delay for Various Functional Units ... 106
Figure 59. Allocating Window Locations without Compressing 107
Figure 60. Performance Effect of Central-Window Size without Compression 107
Figure 61. Performance Effect of Limiting Operand Buses from a Sixteen-Entry Central Window 109
Figure 62. Change in Average Instruction-Issue Distribution from Sixteen-Entry Central Window as Operand Buses are Limited 110
Figure 63. Issuing Loads and Stores with Total Ordering 113
Figure 64. Load Bypassing of Stores 114
Figure 65. Load Bypassing of Stores with Forwarding 116
Figure 66. Performance for Various Load/Store Techniques 117
Figure 67. Reorganizing the Load/Store Unit with a Central Instruction Window 118
Figure 68. Performance for Various Load/Store Techniques using a Central Window 119
Figure 69. Effect of Store-Buffer Sizes 120
Figure 70. Effect of Limiting Address Bits for Memory Dependency Checking ... 121
Figure 7 1. Cumulative Simplifications with Two-Instruction Decoder 126
Figure 72. Cumulative Simplifications with Four-Instruction Decoder 126

Chapter 1
Introduction

The time taken by a computing system to perform a particular application is determined by three factors:

- the processor cycle time,
- the number of processor instructions required to perform the application, and
- the average number of processor cycles required to execute an instruction.

System performance is improved by reducing one or more of these factors. In general, the cycle time is reduced via the implementation technology and memory hierarchy, the number of instructions is reduced via optimizing compilers and software design, and the average number of cycles per instruction is reduced via processor and system architecture.

To illustrate, RISC processors achieve performance by optimizing all three of these factors [Hennessy 1986]. The simplicity of a RISC architecture permits a high-frequency implementation. Also, because the RISC instruction set allows access to primitive hardware operations, an optimizing compiler is able to effectively reduce the number of instructions performed. Finally, a RISC processor is designed to execute almost all instructions in a single cycle. Caches and software pipeline scheduling [Hennessy and Gross 1983] help the processor achieve an execution rate of nearly one instruction per cycle.

In the future, processor performance improvements will continue to result from improving one or more of these factors. The choice among the various techniques to accomplish this is determined by cost and performance requirements of the intended processor application. For example, multi-processing, by providing more than one processor to execute instructions, can reduce by large factors the average number of cycles required for an application, but requires an application that can be decomposed into independent tasks and incurs the cost of multiple processor units.

This thesis is concerned with single-processor hardware architectures that allow a sustained execution rate of more than one instruction per processor cycle: these are called a ***super-scalar*** architectures. Figure 1 is a simple comparison of the pipelines of a ***scalar*** and a super-scalar processor (Figure 1 shows an ideal instruction sequence, where all instructions are independent of each other). The pipeline of the scalar processor can handle more than one

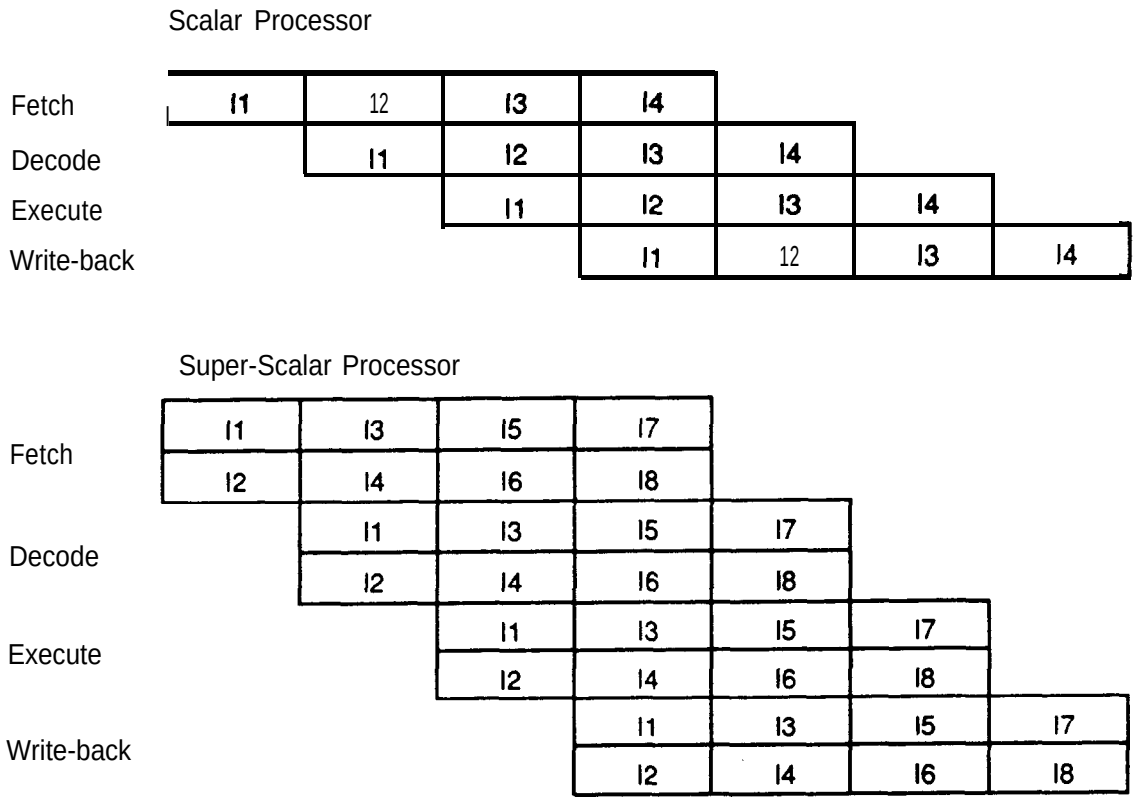

Figure I. Simple Definition of Super-Scalar Processor

instruction per cycle, as long as each instruction occupies a different pipeline stage. However, the maximum capacity of the scalar pipeline (in this example) is one instruction per cycle. In contrast, the super-scalar pipeline can handle more than one instruction both in different pipeline stages and within the same pipeline stage. The capacity of the super-scalar processor pipeline (in this example) is two instructions per cycle.

One way to view an application is that it specifies a set of operations to be performed by processing resources. The efficient execution of these operations is largely a matter of scheduling the use of processor and system resources so that overall execution time is nearly minimum. Processor software and hardware are responsible for generating a schedule that efficiently uses the available resources, and for translating this schedule into actual operations. A software scheduler can arrange the lexical order of instructions so that they are executed in some optimal (or near-optimal) order with respect to efficient use of resources. A hardware scheduler can dynamically arrange the instruction-execution sequence to make

efficient use of resources. In either case, however, the schedule of operations is constrained by data dependencies between instructions and by finite processing resources.

Generating and executing an instruction schedule does not intrinsically depend on the number of instructions that can be performed in a single cycle. The capability to perform more than one instruction per cycle simply makes it possible to more effectively use the available resources than if instruction issue is limited to one instruction per cycle. Whether or not this capability is beneficial depends on scheduling successes of software and hardware.

Most of the published work on instruction schedulers concentrates on specific scheduling algorithms which can be implemented by software or hardware. Software-based studies usually assume minimal processor hardware in a system environment that is constrained to be deterministic and thus permit software scheduling (such as by omitting data caches [Colwell et al. 1987]). On the other hand, hardware-based studies usually assume minimal software support (sometimes even going so far as to claim that hardware scheduling is advantageous because it relieves the compiler of the burden of generating optimized code [Acosta et al. 1986]). Both hardware and software studies typically focus on special-purpose applications, such as vectorizable applications, which are generally easier to schedule than general-purpose applications.

This study uses trace-driven simulation to examine the question of whether the cost, complexity, and performance of hardware scheduling in a super-scalar processor justify its use for general-purpose applications. To address a shortcoming of previous studies, this study examines a wide range of interactions and tradeoffs between processor components, rather than focusing on one specific organization or scheduling algorithm. The approach taken is to evaluate a number of architectural features both in terms of the hardware cost and the performance provided, and to suggest areas where performance is not very sensitive to hardware simplification. As a result, a number of design alternatives are presented. In an actual implementation, design decisions would be further guided by the capabilities and limitations of the implementation technology.

Some of the hardware simplifications suggested in this thesis are the result of software support, but only in cases where software can obviously provide this support. To limit the scope of this study, software techniques are not addressed in detail. Cost andcomplexity are examined in light of simplifications that software might provide, but no software techniques are used to increase performance. The reader interested in software scheduling is referred to: Foster and Riseman [1972], Charlesworth [1981], Fisher [1981], Rau and Glaeser [1981], Fisher [1983], Hennessy and Gross [1983], Mueller et al. [1984], Weiss and Smith [1987],

Hwu and Chang [1988], Lam [1988], Wulf [1988], and Jouppi and Wall [1988]. These references together give an overview of the capabilities and limitations of software schedulers.

This thesis begins by establishing the background of this study. Chapter 2 explains concepts and terminology related to hardware instruction schedulers, and discusses the existing literature. This introductory material motivates the current study.

Chapter 3 introduces the methodology of this study and proposes a super-scalar processor that performs hardware scheduling. With ideal instruction fetching, the proposed processor can realize a speedup of over two for general-purpose benchmarks. However, simple, realistic instruction fetching limits performance to well below a speedup of two.

Chapter 4 explains the causes of the instruction-fetch limitations and techniques for overcoming them. Although instruction-fetch limitations can be largely removed, instruction fetching still limits performance more than the execution hardware. This suggests a design approach of simplifying the execution hardware in light of the instruction-fetch limitations.

Chapters 5 and 6 explore a number of different implementations of the super-scalar execution hardware. Chapter 5 focuses on mechanisms for resolving data dependencies and supplying instructions with operands, and Chapter 6 focuses on hardware instruction scheduling. Both chapters are oriented towards limiting processor hardware without causing a significant reduction in performance.

Finally, Chapter 7 presents the conclusions of this study. A super-scalar processor achieves best performance with a core set of features. These features are complex and interdependent, and the removal of any single feature causes a relatively large reduction in performance. However, there are many possible hardware simplifications in other areas that do not reduce performance very much. These simplifications provide a number of implementation alternatives.

Chapter 2
Background

Hardware instruction-scheduling-both with single-instruction issue and multiple-instruction issue-has been the object of a number of previous investigations. This chapter describes fundamental concepts related to hardware instruction-scheduling, and explores how these concepts have been applied in published research investigations. These investigations form the basis of the current research, either by providing ideas to explore or by indicating fruitless approaches. However, these investigations also leave open a number of questions that are addressed in the current study. Previous studies do not address the effects of super-scalar techniques on general-purpose applications, focusing instead on scientific applications. In addition, they do not address the effects of super-scalar techniques in the context of the compiler optimizations and the low operations latencies that characterize a RISC processor.

2.1 Fundamental Concepts

There are two, independent approaches to increasing performance with hardware instruction scheduling. The first is to remove constraints on the instruction-execution sequence by diminishing the relationship between the order in which instructions are executed and the order in which they are fetched. The second is to remove conflicts between instructions by duplicating processor resources. Either approach, not surprisingly, incurs hardware costs.

2.1.1 Instruction Scheduling Policies

The simplest method for scheduling instructions is to issue them in their original program order. Instructions flow through the processor pipeline much as they do in a scalar processor: the primary difference is that the super-scalar pipeline can execute more than one instruction per cycle. Still, though the super-scalar processor can support a higher instruction-execution rate than the scalar processor, the super-scalar pipeline experiences more operation dependencies and resource conflicts that stall instruction issue and limit concurrency.

Figure 2 illustrates the operation of the super-scalar processor when instructions are issued ***in-order*** and complete ***in-order.*** In this case, the pipeline is designed to handle a certain number of instructions (Figure 2 shows two instructions), and only this number of instructions can be in execution at once. Instruction results are written back in the same order that the corresponding instructions were fetched, making this a simple organization. Instruction

Decode	I1	I3	-	I5			
	I2	I4	-	I6			
Execute		I1	I1	-	-	-	-
		I2	-	I3	I4	I5	I6
Write-back			-	I1	-	I3	-	I5
			-	I2	-	I4	-	I6

Total number of cycles = 8

Notes:
I1 requires two cycles to execute
13 and I4 conflict for functional unit
I5 depends on I4
I5 and I6 conflict for functional unit

Figure 2. Super-Scalar Pipeline with In-Order Issue and Completion

issuing is stalled when there is a conflict for a functional unit (the conflicting instructions are then issued in series) or when a functional unit requires more than one cycle to generate a result.

Figure 3 illustrates the operation of the super-scalar processor when instructions are issued in-order and complete ***out-of-order.*** In this case, any number of instructions is allowed to be in execution in the pipeline stages of the functional units, up to the total number of pipeline stages. Instructions can complete out-of-order because instruction issuing is not stalled when a functional unit takes more than one cycle to compute a result: a functional unit may

Decode	I1	I3	I5				
	I2	I4	I6				
Execute		I1	I1	-	-	-	
		I2	I3	I4	I5	I6	
Write-back			-	I1	-	-	
			I2	I3	I4	I5	I6

Total number of cycles = 7
I1 completes out-of-order

Notes:
I1 requires two cycles to execute
I3 and I4 conflict for functional unit
I5 depends on I4
I5 and I6 conflict for functional unit

Figure 3. Super-Scalar Pipeline with In-Order Issue and Out-of-Order Completion

complete an earlier instruction after subsequent instructions have already completed. Instruction issuing is stalled when there is a conflict for a functional unit, when a required functional unit is not available, when an issued instruction depends on a result that is not yet computed, or when the result of an issued instruction might be later overwritten by an older instruction that takes longer to complete.

Completing instructions out-of-order permits more concurrency between instructions and generally yields higher performance than completing instructions in-order. However, out-of-order completion requires more hardware than in-order completion:

- Dependency logic is more complex with out-of-order completion, because this logic checks data dependencies between decoded instructions and all instructions in all pipeline stages. The dependency logic must also insure that results are written in a correct order. With in-order completion, dependency logic checks data dependencies between decoded instructions and the few instructions in execution (for the purpose of forwarding data upon instruction completion), and results are naturally written in a correct order.

- Out-of-order completion creates a need for functional units to arbitrate for result buses and register-file write ports, because there are probably not enough of these to satisfy all instructions that can complete simultaneously.

Out-of-order completion also complicates restarting the processor after an interrupt or exception, because, by definition, an instruction that completes out-of-order does not modify processor or system state in a sequential order with respect to other instructions. One approach to restart relies on processor hardware to maintain a simple, well-defined restart state that is consistent with the state of a sequentially-executing processor [Smith and Pleszkun 1985]. In this case, restarting after a point of incorrect execution requires only a branch (or similar change of control flow) to the point of the exception, after the cause of the exception has been corrected. A processor providing this form of restart state is said to support ***precise interrupts orprecise exceptions.*** Alternatively, the processor pipeline state can be made accessible by software to permit restart [Pleszkun et al. 1987].

Regardless of whether instructions complete in-order or out-of-order, in-order issue limits performance because there is a limited number of instructions to schedule. The flow of instructions is stalled whenever the issue criteria cannot be met. An alternative is to provide a relatively large set of instructions to be executed-in an ***instruction*** window-from which independent instructions are selected for issue. Instructions can be issued from the window

with little regard for their original program order, so this method can issue instructions ***out-of-or&r.*** Figure 4 illustrates the operation of a super-scalar pipeline with out-of-order issue.

The instruction window is not an additional pipeline stage, but is shown in Figure 4 as a scheduling mechanism between the decode and execute stages for clarity. The fact that an instruction is in the window implies that the processor has sufficient information about the instruction to make scheduling decisions. The instruction window can be formed by somehow looking ahead at instructions to be executed **[Wedig 1982],** or by fetching instructions sequentially into the window and keeping them in the window as long as they cannot be executed.

2.1.2 Scheduling Constraints

Regardless of the sophistication of the scheduling policy, performance is ultimately limited by other constraints on scheduling. These constraints fall into four basic categories:

- ***Procedural dependencies.*** If all instructions to be scheduled were known at the beginning of execution, very high speedups would be possible **[Riseman** and Foster 1972, Nicolau and Fisher **1984]**. Unfortunately, branches cause ambiguity in the set of instructions to be scheduled. Instructions following a branch have a procedural dependency on the branch instruction, and cannot be completely executed until the branch is executed.
- ***Resource conflicts.*** Instructions that use the same shared resources cannot be executed simultaneously.

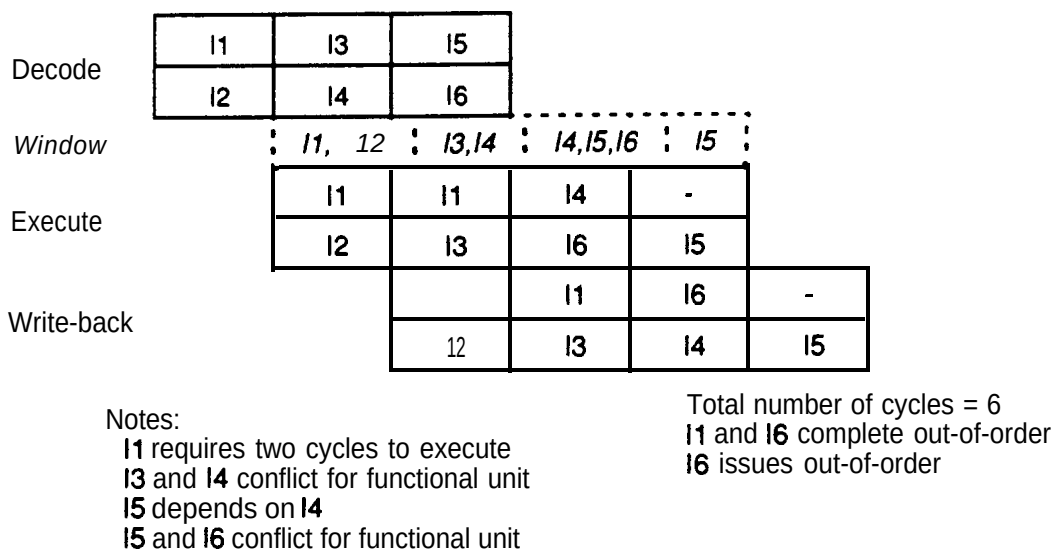

Figure 4. Super-Scalar Pipeline with Out-of-Order Issue and Completion

- ***True data dependencies*** (or ***true dependencies).*** An instruction cannot be executed until all required operands are available.

- ***Storage conflicts.*** Storage locations (registers or memory locations) are reused so that, at different points in time, they hold different values for different computations. This can cause computations to interfere with one another even though the instructions are otherwise independent [Backus 1978]. The term "storage conflict" is not in widespread use, but is more descriptive of the scheduling constraint than other terms in general use. There are two types of storage conflicts: ***anti-dependencies*** and ***output dependencies (there*** is little standardization on the terminology used to denote these dependencies, but the concepts are the same in any case). Anti-dependencies enforce the restriction that a value in a storage location cannot be overwritten until all prior uses of the value have been satisfied. Output dependencies enforce the restriction that the value in a storage location must be the most recent assignment to that location.

Procedural dependencies and resource conflicts are easily understood, and are not discussed further in this section (although they are considered in the remainder of this thesis). True dependencies are often grouped with storage conflicts, in the literature, into a single class of instruction dependencies. However, it is important to distinguish true dependencies from storage conflicts, because storage conflicts can be reduced or removed by duplicating storage locations, much as other resource conflicts can be reduced or removed by duplicating resources.

The distinction between true, anti-, andoutput dependencies is easily illustrated by an example:

```
R3 := R3 op R5     (1)
R4 := R3 + 1       (2)
R3 := R5 + 1       (3)
R7 := R3 op R4     (4)
```

In this example:

- The second instruction cannot begin until the completion of the assignment in the first instruction. Also, the fourth instruction cannot begin until the completion of the assignments in the second and third instructions. The first input operand to the second instruction has a true dependency on the result value of the first instruction, and the input operands to the fourth instruction have true dependencies on the result values of the second and third instructions.

- The assignment of the third instruction cannot be completed until the second instruction begins execution, because this would incorrectly overwrite the first operands of the second instruction. The output result of the third instruction has an anti-dependency on the first input operand of the second instruction.

- The assignment of the first instruction cannot be completed after the assignment of the third instruction, because this would leave an old, incorrect value in register R3 (possibly causing, for example, the fourth instruction to receive an incorrect operand value). The result of the third instruction has an output dependency on the first instruction.

True dependencies are always of concern, because they represent the true flow of data and information through a program. Anti-dependencies are of concern only when instructions can issue out-of-order (or when instructions are reordered by software), because it is only in this situation that an input operand can be destroyed by a subsequent instruction (out-of-order issue can sometimes effectively occur during exception restart in a processor that allows out-of-order completion). Output dependencies are of concern only when instructions can complete out-of-order, because it is only in this situation that an old value may overwrite a more-current value in a register.

Anti- and output dependencies can unnecessarily constrain instruction issue and reduce performance. This is particularly true with out-of-order issue, because these dependencies introduce instruction-ordering constraints that are not really necessary to produce correct results. For example, in the instruction sequence above, the issue of the third instruction might be delayed until the first and second instructions are issued, even when there is no other reason to delay issue.

2.1.3 Storage Conflicts and Register Renaming

In a super-scalar processor, the most significant storage conflicts are caused by the reuse of registers, because registers are most frequently used as sources anddestinations of operands. When instructions issue and complete in order, there is a one-to-one correspondence between registers and values. When instructions issue and complete out-of-order, the correspondence between registers and values breaks down, and values conflict for registers. This problem can be especially severe in cases where the compiler performs register allocation [Chaitin et al. 1981], because the goal of register allocation is to place as many values in as few registers as possible. Having a high number of values kept in a small number of registers

creates a higher number of conflicts when the execution order is changed from the order assumed by the register allocator [Hwu and Chang 1988].

A hardware solution to these storage conflicts is for the processor to provide additional registers which are used to reestablish the correspondence between registers and values. The additional registers are allocated dynamically by hardware, using ***register renaming*** [Keller 1975]. With register renaming, a new register is typically allocated for every new assignment. When one of these additional registers is allocated to receive a value, an access using the original register identifier obtains the value in the newly-allocated register (thus the register identifier is renamed to identify the new register). The same register identifier in several different instructions may access different hardware registers, depending on the locations of register references with respect to register assignments.

With renaming, the example instruction sequence of Section 2.1.2 becomes:

```
R3b := R3. op R5a   (1)
R4b := R3b + 1      (2)
R3c := R5. + 1      (3)
R7b := R3c op R4b   (4)
```

In this sequence, each assignment to a register creates a new ***instance*** of the register, denoted by an alphabetic subscript. The creation of new instances for R4 and R3 in the second and third instructions avoids the anti- and output dependencies on the first and second instructions, and yet does not interfere with correctly supplying operands to the fourth instruction.

Hardware that performs renaming creates each new register instance and destroys the instance when its value is superseded and there are no outstanding references to the value. This removes anti- and output dependencies, and allows more instruction concurrency. Registers are still reused, but reuse is in line with the requirements of concurrent execution.

2.2 Published Techniques for Single-Instruction Issue

This section explores how the concepts and techniques described in Section 2.1 have been addressed by published proposals for hardware scheduling with single-instruction issue. The goal of these proposals is to attempt to achieve an instruction-execution rate of one instruction per cycle with functional units which can take several cycles to produce a result. The primary shortcoming of single-instruction schedulers with regards to the current study is that they (as expected) do not address problems related to multiple-instruction issue. Single-instruction issue greatly reduces the burden on the scheduling logic, because this logic generally deals with instructions one-at-a-time.

2.2.1 Common Data Bus-Tornado's Algorithm

Tomasulo [1967] describes the hardware scheduling algorithm implemented in the IBM 360/91. There are two key components supporting this scheduling algorithm: ***reservation stations*** and a ***common data bus.*** The reservation stations appear at the input of each functional unit, and hold instructions which are waiting on operands or on the availability of the associated functional units. The common data bus carries operands from the output of the functional units to the reservation stations: it is important that all functional units share this common data bus so that operands may be broadcast to all reservation stations.

Tomasulo's algorithm implements register renaming. When an instruction is decoded, it is allocated a reservation-station entry (decoding stalls if there is no available entry). Instruction operands-or tags for these operands, if they are not available-are copied to the reservation-station entry. This copying avoids anti-dependencies, because subsequent instructions can complete and write their results without disrupting either the operands or the tags of this instruction.

To complete instruction decode, the identifier for the allocated reservation-station entry is written into a tag location associated with the result register (if the instruction has a result). This tag identifies the result that will be produced by this instruction, and serves to rename the corresponding register. Any subsequent instruction that is decoded before the current instruction completes obtains this tag instead of an operand value. An instruction is issued from the reservation station when all of its operands are valid (that is, when no tags are held instead of values) and the functional unit is free. When an instruction completes, its result value is broadcast on the common data bus, along with the tag value (in reality, the tag is broadcast on the preceding cycle, for timing reasons). All reservation-station entries that have an operand tag equal to this result tag latch the result value, because this value is required as an operand. The result value may also be written into the register file, but only if its tag matches the tag associated with the register. This avoid output dependencies: if the tags do not match, the result does not correspond to the most recent pending update to the register.

Exceptions are imprecise in this scheme because the sequential-execution state is not maintained in the register file, and there is no provision to recover this state. However, exceptions that are detected during instruction decode (such as invalid instruction codes) are precise. Exceptions detected during decode can be made precise simply by holding instruction decode and allowing all pending instructions to complete.

2.2.2 Derivatives of Tornado's Algorithm

Weiss and Smith [1984] study the performance advantages of Tomasulo's algorithm for the CRAY- 1 scalar processor, using the Lawrence Liver-more Loops as benchmarks. With single-instruction issue, Tomasulo's algorithm yields a 1.58 average speedup. Weiss and Smith also propose and explore two simpler alternative hardware-scheduling mechanisms.

The first alternative to Tomasulo's algorithm is based on Thorton's [1970] register ***scoreboard used*** in the CDC6600. Weiss and Smith examine a variant of Thorton's algorithm that eliminates renaming and the associated tag-allocation hardware of Tomasulo's algorithm. Weiss and Smith's variant of Thorton's algorithm operates in a similar manner to Tomasulo's algorithm, except that a single bit (the scoreboard bit) is used instead of a tag to indicate that a register has a pending update. This eliminates the tag array and replaces it with a simpler scoreboard array. Since there is only one bit to track pending register updates, there can be only one such update, and decoding is stalled if the decoded instruction will update a register that already has a pending update. This further simplifies the hardware, because when there can be only one pending update there is no need to compare result tags to register tags during write-back. Also, register identifiers are used instead of operand tags to match result values to operand values in the reservations stations. The disadvantage of Thorton's algorithm is its performance: Tomasulo's algorithm has 22% better performance for the Livermore Loops, on the average.

A second simplification to Tomasulo's algorithm examined by Weiss and Smith is a mechanism they term ***direct tag search,*** which eliminates the reservation-station comparators for matching result tags to operand tags. Direct tag search is used in a hardware organization very similar to that of Tomasulo's, except that there is a table indexed by result tags to perform the routing function that is performed, in Tomasulo's algorithm, by the reservation-station comparators. In direct tag search, there can be only one reference to a given tagged value in a reservation-station entry; a second attempted reference blocks instruction decoding. When a tag is placed into a reservation-station entry, an identifier for that entry is placed into the tag-table entry indexed by the tag. When the result corresponding to this tag is produced, the tag is used to access the tag table, and the reservation-station identifier in the table used to route the result value to the reservation station. This has about an 8% advantage over Thorton's algorithm, on the average.

Sohi and Vajapeyam [1987] address another undesirable feature of Tomasulo's algorithm: there is a tag entry and comparator for each register, but this hardware is under-utilized. There are not many instructions in the reservation stations at any given time, and thus there

are not many active tags in the tag array. This is particularly a problem with a processor having a large number of registers (Sohi and Vajapeyam are concerned with the CRAY-1 scalar processor, which has 144 registers). As an alternative, they propose keeping all active tags in a much smaller associative array- the *rag* unit-that maintains all register-to-tag mappings currently in effect. When an instruction is decoded, and one or both of its source registers have pending updates (indicated by a single bit accessed with the source registers), the tag unit is queried with source-register identifiers and returns the appropriate tag(s). At the same time, a new tag is allocated for the current result, and this is placed into the tag unit along with the destination-register identifier. If there is another pending update for the same register in the tag array, this previous update is marked as not being the latest update, to avoid output dependencies.

Sohi and Vajapeyam [1987] also examine several extensions to the tag unit. The first extension follows the observation that each tag entry in the tag unit is associated with a single instruction in a reservation station. Because of this one-to-one correspondence between tag-unit entries and reservation-station entries, the tag unit and reservation stations can be combined into a single *reservation station/tag unit* (RSTU) that serves as a reservation station for all functional units. This has the advantage that less storage is needed, because the reservation station is not partitioned by functional unit. Furthermore, the storage in the RSTU can be used to hold results after instruction completion, and can return the results to the register file in sequential order. This configuration, called a *register update unit* (RUU), provides precise interrupts. The RW is operated as a first-in, first-out (FIFO) buffer, with decoded instructions being placed at the tail of the FIFO and results being written from the head of the FIFO.

Updating the registers sequentially allows variations in the tag-allocation hardware. If results are returned to registers in order, counters can be used to keep track of pending updates. Sohi and Vajapeyam associate two counters with each register: one counter keeps track of the number of values destined for the register (the number of *instances* counter), and the other keeps track of the most recent value *(the latest instance* counter). The latest instance counter-appended to the register identifier—acts as a tag for the operand. The number of instances counter prevents the number of pending updates from exceeding the limitations of the latest instance counter and thus prevents duplication of tags (this could be accomplished more simply by making the latest instance counter big enough to count every entry in the RW). It is not clear that these counters are much simpler than the tag logic they are intended to avoid.

Sohi and Vajapeyam find that their configuration has a speedup of 1.81 for the Lawerence Livermore Loops with a register update unit of 50 entries. The CRAY-1 configuration they use has high operation latencies which reduce the advantage of hardware scheduling.

2.3 Published Techniques for Multiple-Instruction Issue

This section explores how the concepts and techniques described in Section 2.1 have been addressed by published proposals for hardware scheduling with multiple-instruction issue. In contrast to single-instruction schedulers, multiple-instruction schedulers are in a more theoretical domain. Most of the proposals discussed below concentrate on scheduling algorithms and are light in their treatment such issues as efficient instruction fetching, the complexity and speed of the dependency-checking logic, and the allocation and deallocation of processor resources when a variable number of instructions issue and complete in a single cycle.

2.3.1 Detecting Independent Instructions with a Pre-Decode Stack

The original work on hardware scheduling for multiple-instruction issue appears to have been by Tjaden and Flynn [1970]. They describe a method for detecting independent instructions *in* a *pre-decode stack,* considering only data dependencies between instructions. Branch instructions are not considered: it is assumed that procedural dependencies have been removed before the instructions are placed into the pre-decode stack.

In Tjaden and Flynn's proposal, instructions are encoded so that instruction source and sink registers are identified by bit vectors within the instructions-it is contemplated that the processor has only a few such sources and sinks, as in an accumulator-based architecture. Dependencies between instructions in the pre-decode stack are determined by comparing these source and sink vectors bit-by-bit on every cycle, causing the dependency-checking logic to grow as the square of the size of the pre-decode stack. The algorithm checks for true and anti-dependencies, but completely ignores output dependencies. Apparently, it is assumed that a value will be used as a source operand before it is overwritten, and that all instructions complete in the same amount of time. The algorithm also defines *weakly independent* instructions which use different registers but which may generate a common memory address and thus depend through memory locations.

The scheduling algorithm implements renaming by a special treatment of instructions having *open effects*. An instruction has open effects if it updates a register on which it does not depend-this is simply the property an instruction must have for renaming to be

advantageous. To support renaming, each register is statically duplicated a number of times, so that each program-visible register is actually implemented as a vector of registers. When an open-effects instruction is found in the pre-decode stack, its destination register is reassigned to be the next adjacent register in the register vector, all instructions following this instruction then use the newly-assigned register, until a new open-effects instruction is encountered which updates this register.

Tjaden and Flynn find that, with renaming, the algorithm has a potential speedup of 1.86. However, this measurement was made for only 5600 instructions, and is based only on the number of independent instructions in the pre-decode stack. This speedup does not account for effects of instruction fetching, instruction-execution time, nor increased decoding time.

2.3.2 Ordering Matrices

Tjaden [1972] and Tjaden and Flynn [1973] describe a (rather theoretical) scheduling technique for issuing multiple instructions based on software-generated dependency matrices. Each row or column of a dependency matrix is a bit vector that identifies the sources or sinks of instructions in a program. There are two principal matrices. The first matrix has rows made up of the bit vectors that identify instruction sinks: a bit is set in a row if the instruction corresponding to the row alters the register corresponding to the bit. The second matrix has columns made up of bit vectors that identify instruction sources: a bit is set in a column if the instruction corresponding to the column reads the register corresponding to the bit. Dependencies between instructions are detected by performing operations on these matrices. As in Tjaden and Flynn [1970], it is unclear how output dependencies are handled.

The dependency matrices are assumed to be computed by a software preprocessor and loaded into the processor before execution. For this reason, the algorithm cannot handle dependencies that are detected via dynamically-computed addresses. Furthermore, elements of the matrix are activated and deactivated by hardware to enable and disable portions of the dependency-checking as instructions are completed and reactivated (reactivation occurs because of branches). Because of activation and deactivation, matrix elements can have three values: off (the source or sink is not used), on/active (the source or sink will be used), and on/inactive (the source of sink will not be used because the instruction is not active, but may be used at a later time). The algorithm handles procedural dependencies by making this activation/deactivation information explicitly visible as storage, called the *IC* resources. All branch instructions update the IC resources to activate and deactivate instructions, and all instructions use the IC resources as a source. Procedural dependencies are thus treated as data dependencies on the IC resources. This causes all instruction preceding a branch to

issue before the branch is issued, and causes a branch to issue before instructions after the branch are issued (although the algorithm can be enhanced in several ways to relax these constraints).

Register renaming is performed as in Tjaden and Flynn [1970] by providing statically-duplicated registers. In this case, however, renaming is applied to instructions having shadow *effects*. The term shadow effects refers to an instruction with an antidependency on a previous instruction: this instructions can be made independent by reassigning the instruction sink. Renaming requires five-value matrix elements: off, on/active, on/inactive, shadow effects/active (the instruction is independent if the sink can be reassigned), and shadow effects/inactive. Handling shadow effects in this manner has the further disadvantage of creating false dependencies, because the dependency matrices do not take renaming into account.

The speedup for the highest-performing version of this algorithm is 1.98 on a sample of three benchmarks. However, this speedup does not account for the complexity of the scheduling algorithm. An implementation with 36-by-36 matrices requires 1296 total elements in each matrix, and 3 bits of storage at each element. Performing the required matrix operations and updates takes 4 micro-seconds, assuming six-input gates with a propagation delay of 5 nanoseconds each. Even at this, the algorithm can handle only tasks that are 36 instruction long or less. Larger tasks are handled by hierarchically breaking the tasks into smaller sub-tasks and using the algorithm to schedule these sub-tasks.

2.3.3 Concurrency Detection in Directly-Executed Languages

Wedig [1982] builds on the the work Tjaden [1972] and Tjaden and Flynn [1973], addressing primarily the weaknesses in handling procedural dependencies and in fetching tasks into the instruction window. Wedig's scheduling algorithm assumes instructions are in their original static order in the instruction window. This algorithm is presented in the context of a directly-executed-language architecture, but this limitation seems unnecessary other than to allow some assumptions about how instruction and task information are encoded.

Wedig assumes the existence of a resource ***conflict*** function returning a set of independent instructions (this function has the complexity of Tjaden's and Flynn's ordering matrices). Procedural dependencies are handled by associating with each instruction in the window a execution vector that indicates how many iterations of the instruction have been executed. A global ***to-be-executed*** element defines the total number of iterations to be performed by all instructions in the window. This approach initially assumes that all instruction in the window will be performed the same number of times (conceptually equal to the highest number

of iterations in the window). Instructions that are not actually executed this number of times *are virtually executed* by updating their execution vector without actually executing the instruction. The execution vector and the to-be-executed vector are computed dynamically as the result of branch instructions. The static instructions in the window can thus provide many dynamic instructions. Several branches can be executed in parallel, because these simultaneously update the to-be-executed vectors. Wedig also examines the problems related to filling the instruction window, but this amounts to little more than enumerating the problems encountered and defining the functions that must be performed to update the execution structures.

Wedig implements register renaming by associating a shadow *storage queue with* each instruction in the window. The shadow storage queue consists of three vectors. There is one *shadow sink vector;* an element of this vector is the value produced by the corresponding iteration of the instruction, and there is one element per iteration (over a local span of time). There are two shadow *source vectors;* elements of these vectors point to source operands, in the shadow sink vectors, of the corresponding instruction iterations. The shadow storage queue unnecessarily duplicates resources, and further complicates the scheduling algorithm. With renaming, Wedig's algorithm achieves a speedup of 3.00, assuming compiler support to help create independent instructions.

Uht [1986] builds on Wedig's work, and reduces hardware required by building dependency matrices as instructions are loaded into the window rather than after instructions are loaded into the window. Uht also identifies classes of procedural dependencies that are avoided by the static instruction window (for example, nested forward branches can be executed independently, because the execution of these involves only updating the to-be-executed vectors). With renaming, Uht obtains an average speedup of 2.00 over ten benchmarks.

2.3.4 Dispatch Stack

Tomg [1984] and Acosta et *al.* [1986] propose a *dispatch stack* that detects independent instructions and issues them to the functional units. The dispatch stack is an instruction window that has source- and destination-register fields augmented with dependency counts. There is a dependency count associated with each source register, giving the number of pending updates to the source register (thus the number of updates which must be completed before all possible true dependencies are removed). There are two similar dependency counts associated with each destination register, giving both the number of pending uses of the register (the number of anti-dependencies) and the number of pending updates to the register (the number of output dependencies). When an instruction is loaded into the dispatch

stack, these counts are set by comparisons of register identifiers with all instructions already in the dispatch stack (requiring five comparators per loaded instruction per instruction in the dispatch stack). As instructions complete, the dependency counts are decremented (by a variable amount) based on the source- and destination-register identifiers of completing instructions (also requiring five comparators per completed instruction per instruction in the dispatch stack). An instruction is independent when all of its counts are zero, and can be issued if it has no functional-unit conflicts.

The dispatch stack achieves a speedup of 2.79 for the Lawrence Livermore Loops, on an ideal processor that has an infinite number of single-cycle functional units and infinite instruction-fetch bandwidth. The performance of this machine could be much higher: performance suffers because this algorithm does not implement register renaming. This is unfortunate, because the comparators used to detect dependencies and to update dependency counts could be put to better use by implementing renaming. Also, a branch is handled by stalling the instruction fetcher until the branch is resolved. This apparently does not affect performance very much, because in the Livermore Loops the branch dependency path is shorter than other dependency paths. Dwyer and Tomg [1987] address issues of hardware complexity and performance with the dispatch stack, relying on bit vectors to select registers.

2.3.5 High Performance Substrate

The High Performance Substrate (HPS) [Patt et al. 1985a] and the proposed single-chip implementation HPSm [Hwu and Patt 1986] are concerned with scheduling micro-instructions that have been decoded from higher-level instructions. The motivation for the word "substrate" in the project name is that this architecture can form the basis for emulating any instruction set-via micro-code interpretation—with presumed good performance because of the ability to exploit micro-code-level concurrency.

In HPS, a "branch predictor" supplies instructions to a decoder (the most explicit discussion of the branch predictor appears in [Patt et al. 1985b], but this is not very specific). The decoder then expands instructions into sets of ***nodes. The*** term "node" refers to a node in a dependency sub-graph for the instruction. A node is basically a micro-instruction, and the nodes (microinstructions) corresponding to a given instruction have interdependencies explicitly identified. After decoding, nodes are ***merged*** into a node table using Tomasulo's algorithm The merge operation resolves dependencies on nodes outside of the decoded set, supplying either values or tags for required operands; a ***registrer alias table*** implements renaming. After merging, the nodes are placed into a ***node table*** which serves the same role as

reservation stations, except that there is a single node table for all functional units instead of a node table per functional unit. Nodes are issued from the node table to the functional units. A checkpoint mechanism deals with exceptions and repairing processor state after mispredicted branches [Hwu and Patt 1987].

The proposed HPS hardware is quite complex and general, addressing several concerns not directly related to scheduling or concurrent execution. For example, interpreting higher-level instructions creates a need to identify the point at which all micro-operations for a given instruction have completed, because only at this time is the entire instruction complete.

Hwu and Patt [1986] compare the performance of HPSm to the Berkeley RISC II processor. The performance of HPSm is better than the performance of RISC II, but this is mostly the result of assuming a 100 nano-second cycle time for HPSm against a 330 nanosecond cycle time for RISC II (this is justified on the basis of pipelining). On a cycle-by-cycle basis, HPSm is slower than RISC II in four of six benchmarks, with optimized code.

2.3.6 Multiple-Instruction Issue with the CRAY-1 Architecture

Plezkun and Sohi [1988] describe the performance of the CRAY-1 scalar architecture on the Lawrence Liver-more Loops. They simulate several combinations of result-bus organizations, issue policies (in-order, out-of-order), and operation latencies. Register renaming, with the register update unit of Sohi and Vajapeyam [1987], is also examined. None of these alternatives achieves an instruction-execution rate greater than unity on scalar code, but this is probably due to the high latency of the functional units and of the memory (a load requires eleven or five cycles to execute, and a branch requires five or two cycles, depending on the machine organization). The high latency is reflected in the fact that, for this organization, the maximum theoretical execution rate is .79–1.29 instructions per cycle, Issuing four instructions per cycle with renaming and out-of-order execution achieves 6449% of this maximum. However, single-instruction issue with renaming and out-of-order execution achieves 5662% of the theoretical maximum-close to the performance with multiple-instruction issue. Limitations on performance that are unrelated to the scheduling mechanism make it difficult to determine the effectiveness multiple-instruction scheduling in this case.

2.4 Observations and Conclusions

Much of the previous research is not directly relevant to a super-scalar RISC processor, for two reasons. First, this research was based on architectures with high operation latencies-

or, worse, unstated operation latencies-which blurs the relationship between instruction scheduling and performance. A RISC processor is typically characterized by very low latencies for most operations. Second, the effect of compiler optimizations is often ignored in previous studies, even though these optimization can reduce instruction independence [Jouppi and Wall 1988]. For example, moving an invariant instruction out of a loop can eliminate many independent run-time operations.

Previous studies have focused on scientific applications having a high degree of independent computation. Simply having the capability to issue floating-point instructions concurrently with other instructions gives the super-scalar processor the same high instruction throughput, on vectorizable code, as a vector processor. In some scientific applications, a super-scalar processor can be superior to a vector processor, because the super-scalar processor does not incur as much of the pipeline-startup latency associated with the vector processor [Agerwala and Cocke 1987].

In contrast, the current study is oriented towards applications having widespread use on general-purpose, single-chip RISC microprocessors. These applications are significantly different than scientific applications. There is less opportunity for concurrent execution of operations than there is in scientific programs. Furthermore, hardware costs are a much larger consideration in general-purpose computing than in high-performance scientific computing.

Finally, previous studies provide many ideas for implementing an effective execution unit, but say very little about supplying instructions to the execution unit at an adequate rate. This study will show that instruction fetching is the most severe performance limit in the super-scalar processor, and that the design of the execution unit should take this into account.

Chapter 3
Methodology and Potential Performance

Prior hardware-scheduling studies are a good source of i&as, but these studies do not provide much information on which to base implementation decisions. All research described in the previous chapter concentrates on specific scheduling algorithms, and evaluate implementation tradeoffs only within the context of the given algorithm. The current study, in contrast, explores a wide range of design tradeoffs presented by a super-scalar processor, using highly-optimized, general-purpose benchmark programs. The methodology was designed to allow the efficient evaluation of a large number of hardware alternatives on a large class of programs.

This chapter introduces the methodology of this research. This chapter also develops, as a basis for investigation, a super-scalar RISC processor model used to evaluate potential performance. The performance achieved by this processor model-for perfectly-predicted branches and infinite instruction bandwidth-is over twice the performance of a scalar processor that issues single instructions in order. This speedup is limited by data dependencies, resource conflicts, and the limited number of instructions available to the hardware scheduler. However, as will be shown, instruction-fetch limitations constrain the performance of the super-scalar processor more than instruction-scheduling constraints. The following chapter considers methods for improving the efficiency of the instruction fetcher.

3.1 Simulation Technique

This study uses a trace-driven simulation to evaluate the benefits of various super-scalar organizations for highly-optimized, non-scientific applications. The tracing system is based on the MIPS, Inc. R2000™ RISC processor (1). The R2000 has very good optimizing compilers, as well as analysis tools which allow generation of dynamic instruction traces and data-reference traces [MIPS 86].

All simulations are performed as shown in Figure 5. The optimized object code for a particular benchmark is processed by pixie™ (1) [MIPS 86], a program that annotates the object code at basic-block entry points and at memory references. When the annotated code is executed, it produces a dynamic trace stream in addition to the program's normal output. This trace stream consists of basic block addresses, counts of the number of instructions in

(1) R2000 and pixie are trademarks of MIPS, Inc.

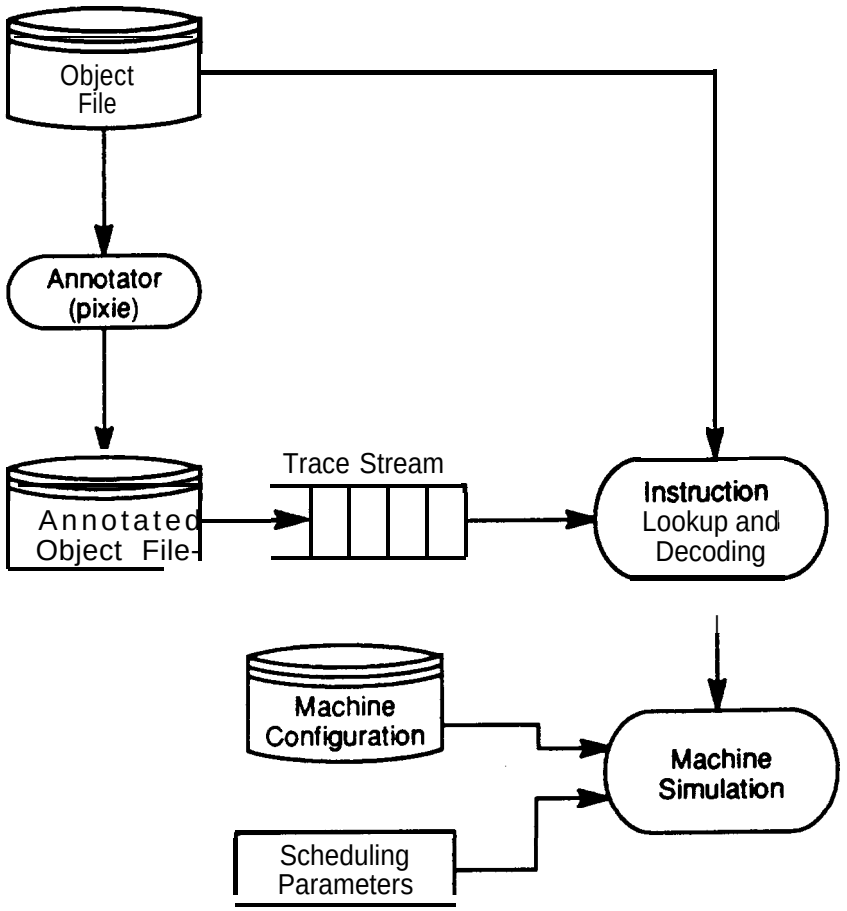

Figure 5. Flowchart for Trace-Driven Simulation

each basic block, and load/store addresses. The trace stream and the original object file are used to generate the full instruction trace stream needed by the hardware simulator. This approach allows statistics to be generated rapidly, and allows the super-scalar evaluation to be performed at a reasonable level of abstraction. For example, the simulator need not track the contents of every processor register, it is only necessary that the simulator be aware of register identifiers in instructions. Furthermore, the simulator need only model the functional units as nodes representing delay.

The machine simulator models the functionality of both a super-scalar and a scalar processor using a machine configuration file and command-line scheduling parameters. The execution of the scalar and super-scalar processors are modeled at a functional level, with time recorded in terms of machine cycles. The cycle time and functional-unit characteristics of the two processors are identical, although the super-scalar processor has two integer arithmetic/logic units (ALUs), which reduces the principle resource constraint to concurrent

instruction execution. Most instructions in the benchmark programs (typically 50%) are performed by an integer ALU, and providing two ALUs relieves the most severe resource bottleneck [Smith et al.1989].

The scalar and super-scalar processors differ primarily in the instruction decode and scheduling mechanisms. The scalar processor issues instructions in-order and completes instructions out-of-order, this permits a certain amount of concurrency between instructions (for example, between loads and other integer operations). The super-scalar processor is able to decode and issue any number of instructions, using any issue policy (the actual processor configuration is specified upon program initiation). By keeping the number of instructions completed by each processor equal, the simulator is able to determine the performance advantage of the super-scalar organization over the scalar processor in a way that is independent of other effects, such as changes in cache reload penalties or functional-unit latencies.

As the simulator fetches instructions from the dynamic trace stream, it accounts for penalties which would be associated with fetching these instructions in an actual implementation (such as instruction-cache misses) and supplies these instructions to the scalar and super-scalar processor models. Each processor model accounts for penalties caused by branches and by the execution unit not being able to accept decoded instructions (because of resource conflicts, for example). Instructions are checked for data and/or register dependencies on all other uncompleted instructions. When an instruction is issued, it flows through a simulated functional-unit pipeline. The instruction cannot complete until it successfully arbitrates for one of (possibly) several result buses. During the cycle in which the result appears on a result bus, all dependencies on the instruction are removed.

3.2 Benchmark Programs

The set of target applications has a significant influence on the available instruction independence and the resulting performance of the super-scalar processor. A significant portion of the published research strives to use super-scalar techniques to enhance the performance of scientific or vectorizable applications. These applications typically have a high number of independent operations and a high ratio of computation to dynamic branches. This in turn leads to large potential concurrency. In contrast, the programs used in this study mostly represent a wide class of general, non-vectorizable applications (linpack is a notable exception). These applications are thought to have little instruction-level concurrency because of their complex control flow, and they form a good test of the generality of the super-scalar processor.

The benchmarks include a number of integer programs (such as compilers and text formatters) and scalar floating-point applications (such as circuit and timing simulators). Table 1 describes each benchmark program. To avoid examining concurrency that is due to the lack of compiler optimization, highly-optimized versions of these programs are used. No software reorganization is performed, except for usual RISC pipeline scheduling already existing in the code.

To obtain processor measurements, each program is allowed to complete 4 million instructions. This reduces simulation time, and includes a fixed overhead for cache cold-start effects in every measurement (roughly corresponding to 100–300 process switches per second, depending on the cycle time and execution rate of a particular configuration). In practice, the results do not change significantly after the first million instructions.

Four programs in Table ***l-ccom, irsim, troff,*** and ***yacc-are*** chosen as a representative sample for demonstrating and clarifying certain points throughout this thesis. This sample

Table l. Benchmark Program Descriptions

Program	Description
5diff	text file comparison
awk	pattern scanning and processing
ccom	optimizing C compiler
compress	file compression using Lempel-Ziv encoding
doduc	Monte-Carlo simulation, double-precision floating--point
espresso	logic minimization
gnuchess	computer chess program
grep	reports occurrences of a string in one or more text files
irsim	delay simulator for VLSI layouts
latex	document preparation system
linpack	linear-equation solver, double-precision floating-point
nroff	text formatter for a typewriter-like device
simple	hydrodynamics code
spice2g6	circuit simulator
troff	text formatter for typesetting device
wolf	standard-cell placement using simulated annealing
whetstone	standard floating-point benchmark, double-precision floating-point
vacc	compiles a context-free arammar into LR(1) parser tables

limits the amount of data presented for purposes of illustration. Othenvise, simulation results are presented as low, harmonic mean, and high values for all programs shown in Table 1. The harmonic mean is used to mote closely convey the expected speedup over all benchmarks.

3.3 Initial Processor Model

Super-scalar processor design presents a large number of opportunities for tradeoffs. As with any processor, there is much interaction between processor components, and this interaction limits performance. However, the interaction is significantly more complex in a super-scalar processor than in a scalar RISC processor. The investigation of tradeoffs is simplified if there is a single basis for evaluation.

This section presents a model for a super-scalar execution unit that was culled from a variety of hardware instruction-scheduling proposals. This model resulted from an investigation into the performance limitations imposed by the execution hardware, and is a good starting point for this study because the model is rather ambitious and can exploit much of the concurrency available in the benchmark programs. Since this execution model does not limit performance appreciably, it allows a more straightforward exploration of performance limits and design tradeoffs than would a processor having limited performance potential.

To develop this initial model, it was necessary to specify criteria for selecting among alternative hardware organizations--in the absence of specific measurements-because of the variety of existing super-scalar proposals and associated performance claims. This section describes the rationale behind the initial processor organization as well as the organization itself. It should be emphasized that the rationale presented here is not necessarily universal: it simply introduces the considerations related to multiple-instruction issue and to hardware scheduling, and motivates the hardware organization. The following chapters deal with exploring, refining, and simplifying this model.

3.3.1 Basic Organization

Three organizational techniques allow the super-scalar processor to achieve as much instruction-execution concurrency as possible. First, there are multiple, independent, functional units. This allows concurrent operation on different instructions. Second, the functional units that take more than one cycle to generate a result are pipelined where possible, so that instructions can be issued at a higher rate than without pipelining. Third, the execution

unit can issue instructions out-of-order to allow the functional units to be better utilized in the presence of data dependencies.

Figure 6 illustrates a cost/performance dilemma presented by a super-scalar processor. Figure 6 plots-for the sample **benchmarks**—the fraction of total cycles in which 0, **1, 2, 3,** etc. instructions are issued in a processor with perfect instruction and data caches (a perfect cache is infinite and suffers no reload penalty). Figure 6 also shows the net instruction-execution rate for these benchmarks. These data illustrate that the maximum number of instructions issued in one cycle can be significantly higher than the aggregate instruction-issue rate (more than a factor of two higher). Hardware costs and performance are determined by the peak instruction-execution rate, but the average execution rate is well below below this peak rate.

To avoid as many constraints as possible, the execution unit must support a high peak instruction-execution rate. However, issuing many instructions per cycle can be quite expensive, because of the cost incurred in communicating operand values to the functional units. Each instruction issued in a single cycle must be accompanied by all its required operands, so issuing N instructions in one cycle requires access ports and routing buses for as many as 2N operands. An apparently cost-effective way to support this peak instruction-issue rate is

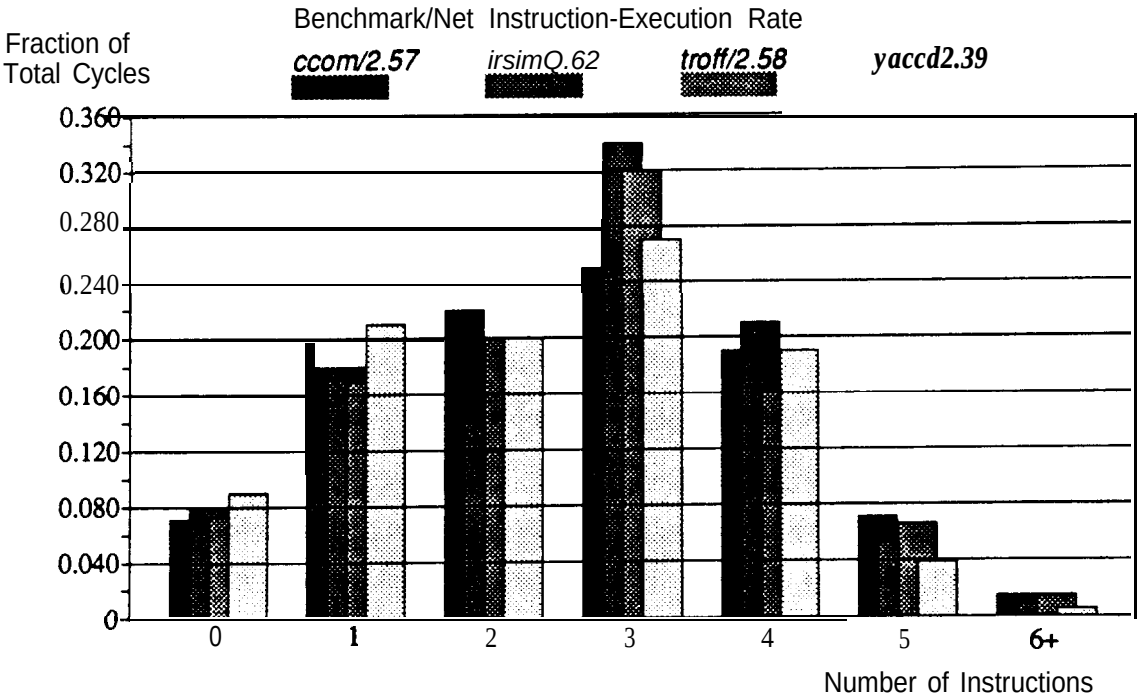

Figure 6. Sample Instruction-Issue Distribution

by distributing the instruction window among the functional units as reservation stations [Tomasulo 1967]. With this organization, the reservation stations act as an instruction buffer that is filled at the average instruction-execution rate. The decoder, register read ports, and operand-distribution buses need only support the average rate. Because the reservation stations are distributed, they can easily support the maximum instruction-issue rate: in one cycle, the reservation stations can issue an instruction at each of the functional units. The data needed for instruction issue comes from the local reservation stations, rather than a global register file.

3.3.2 Implementation Description

Figure 7 shows a block diagram for the processor model. In Figure 7, connections between components should not be construed as single buses; almost all connections comprise multiple buses, as described later.

The processor incorporates two major operational units-an integer unit and a floating-point unit-which communicate data via a data cache and data memory. This organization was chosen to mimic the organization of the R2000 processor and the R2010™ (2) floating-point unit. The integer and floating-point units are controlled by a single instruction stream supplied by an instruction memory via an instruction cache. Both the instruction memory and the instruction cache can supply multiple instructions per fetch, but the number of instructions fetched from the instruction memory is not necessarily equivalent to the number of instructions fetched from the instruction cache.

Each operational unit is composed of a number of functional units with associated reservation stations. A multiple-instruction decoder places instructions and operands into the reservation stations of the appropriate functional units. A functional unit can issue an instruction in the cycle following &ode if that instruction has no data dependencies and the functional unit is not busy; otherwise, the instruction is stored in the reservation station until all dependencies are released and the functional unit is available.

Register renaming [Keller 1975] eliminates anti- and output dependencies. To implement register renaming, each operational unit incorporates a reorder buffer [Smith and Pleszkun 1985] containing a number of storage locations which are dynamically allocated to instruction results. When an instruction is decoded, its result value is assigned a reorder-buffer location, and its destination-register number is associated with this location (i.e. the destination register is renamed). A tag is allocated for the result, and this is also stored in the

(2) R2010 is a trademarks of MIPS, Inc.

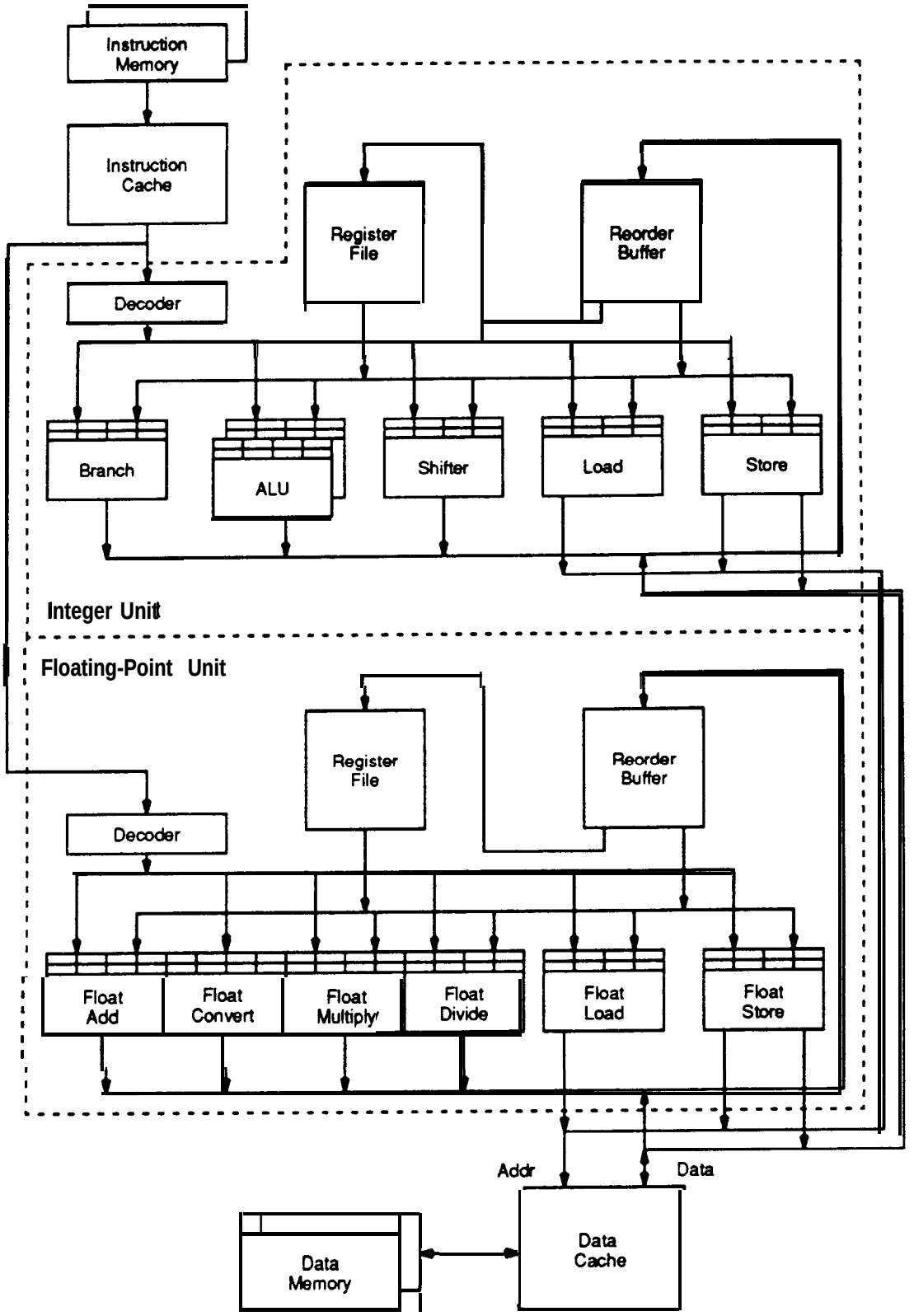

Figure 7. Block Diagram of Processor Model

assigned reorder-buffer location. A subsequent reference to the renamed destination register obtains the value stored into the reorder buffer or the tag for this value.

The reorder buffer is implemented as a content-addressable memory. It is accessed using a register number as a key, and returns the latest value written into the register. This organization performs name mapping and operand access in a single cycle, as does the register file. During instruction decode, the reorder buffer is accessed in parallel with the register file. Then, depending on which one has the most recent value, the desired operand is selected. The operand value-if available-is copied to the reservation station. If the value is not available (because it has not been computed yet), the result tag is copied to the reservation station. This procedure is carriedout for each operand required by each decoded instruction.

If a result register mapped by the reorder buffer is the destination of a previously-decoded instruction, the previous mapping is marked as invalid, so that subsequent instructions obtain the result of the new instruction. At this point, the old register value could be discarded and the reorder-buffer entry freed. However, the entry is preserved in this model because the entry allows the processor to recover the state associated with in-order completion, simplifying interrupt and exception handling [Smith and Pleszkun 1985].

When a result becomes available, it is written to the reorder buffer and to any reservation-station entry containing a tag for this result (this requires associative memory in the reservation stations). Subsequent instructions continue to fetch the value from the reorder buffer—unless the entry is superseded by a new value-until the value is retired by writing it to the register file. Retiring occurs in the order given by sequential execution, preserving the sequential state for interrupts and exceptions.

The design of the super-scalar pipeline closely parallels the design of a scalar RISC pipeline, to keep the execution rate of sequential instructions as high as their execution rate in a scalar processor. Table 2 shows the parallels between the pipeline of the sequential processor and the pipeline of the super-scalar processor. Table 2 does not show every function performed in the pipeline stages, but does illustrate how the additional functions required by the super-scalar processor fit into the pipeline stages of the sequential processor. In essence, the reorder buffer augments the register file and operates in parallel with it. The reservation stations replace the functional-unit input latches. The distribution of operands and writing of results are similar for both processors, except that the super-scalar processor requires more hardware, such as buses and write ports.

Loads and stores occur on a single-word interface to a data cache (there are separate buses for addresses and data). Loads are given priority to use the data-cache interface, since an

Table 2. Comparisons of Scalar and Super-Scalar Pipelines

Pipeline Stage	Scalar Processor	Super-Scalar Processor
Fetch	fetch one instruction	fetch multiple instructions
Decode	decode instruction access operand from register file copy operands to functional-unit input latches	decode instructions access operands from register file and reorder buffer copy operands to functional-unit reservation stations
Execute	execute instruction	execute instructions arbiirate for result buses
Write-back	wriie result to register file forward results to functional-unit input latches	write results to reorder buffer forward results to functional-unit reservation stations
Result Commit	n/a	write results to register file

uncompleted load is more likely to stall computation. Stores are buffered to resolve contention with loads over the data-cache interface. A load is issued in program-sequential order with respect to other loads, and likewise for a store. Furthermore, a store is issued only after all previous instructions have completed, to preserve the processor's sequential state in the data cache (the store buffer also aids in deferring the issue of stores).

Dependencies between loads and stores are determined by comparing virtual addresses. For this study, it is assumed that there is a unique mapping for each virtual page, so that virtual-address comparisons detect all dependencies between the physical memory locations addressed by a process. A load is held because of a store dependency if the load address matches the address for a previous store, or if the address of any previous store is not yet valid. If the valid address of a store matches the address of a load, the load is satisfied directly from the store buffer-once the store data is valid-rather than waiting on the store to complete.

3.3.3 Processor Configuration

For the results presented in Section 3.4, the characteristics of the functional units are modeled after the R2000 processor and R2010 floating-point unit. Table 3 shows the configuration of the functional units. Issue *latency* is defined as the minimum number of cycles

Table 3. Configuration of Functional Units

Fundional Unit	Issue Latency (cycles)		Result Latency (cycles)		Reservation Station Entries
	single	double	single	double	
Integer ALU (2)	1	n/a	1	n/a	4
Barrel Shifter	1	n/a	1	n/a	2
Branch Unit	1	n/a	n/a	n/a	4
Load Unit	1	2	2	3	8
Store Unit	1	2	n/a	n/a	8
Float Add	1	2	2	3	2
Float Multiply	1	2	4	6	2
Float Divide	12	27	12	27	2
Float Convert	1	2	2	4	2

between the issuing of two instructions to a functional unit. ***Result latency*** is the number of cycles taken by the functional unit to generate a result value. Issue and result latencies can depend on whether an operand is a single (32-bit) word or a double-precision (64-bit) floating-point number. Table 3 also shows the number of reservation-station entries allocated to each of the functional units.

The instruction and data caches of both the scalar and the super-scalar processors are 64 kilobyte (Kbyte), direct-mapped caches with a four-word block size. The instruction and data caches are loaded via a double-word interface to the main memory; after an initial access time of twelve cycles (chosen in light of the anticipated processor operating frequency), the interface can reload an entire cache block in two cycles. The integer reorder buffer has sixteen entries, the floating-point reorder buffer has eight entries, and the store buffer has eight entries. Both the integer and the floating-point reorder buffers can accept two results per cycle, and can retire two results per cycle to the corresponding register file. The integer and floating-point reorder buffers and register files have a sufficient number of read ports to satisfy all instructions decoded in one cycle.

Again, this processor configuration is intended only as a basis for further study. The hardware-and the size of the reservation stations-were determined by experimentation to not limit performance significantly for the target applications. This processor organization is used as a standard for comparing other hardware alternatives throughout this study.

3.4 Results Using an Ideal Instruction Fetcher

Figure 8 shows the performance of the execution hardware described in Section 3.3 on the benchmark programs. In Figure 8, performance of the three scheduling policies described in Sec tion 2.1.1 is measured in terms of ***speedup***, where **speedup** is the total number of cycles taken by the scalar processor to execute a benchmark divided by the total number of cycles taken by the super-scalar processor for the same benchmark. The results in Figure 8 are obtained with ideal instruction fetching. The instruction-fetch unit can supply as many instructions in one cycle as the execution unit can consume, except when there is an **instruction-**cache miss. The fact that the simulator is trace-driven allows it to supply instructions for scheduling with little regard for branches. Of course, this ability is not available to any real processor.

With in-order issue, out-of-order completion yields higher performance than in-order completion. In fact, in-order completion has lower performance than the scalar processor for some floating-point-intensive benchmarks. Using out-of-order completion, a scalar processor can achieve more instruction concurrency in the presence of high-latency operations

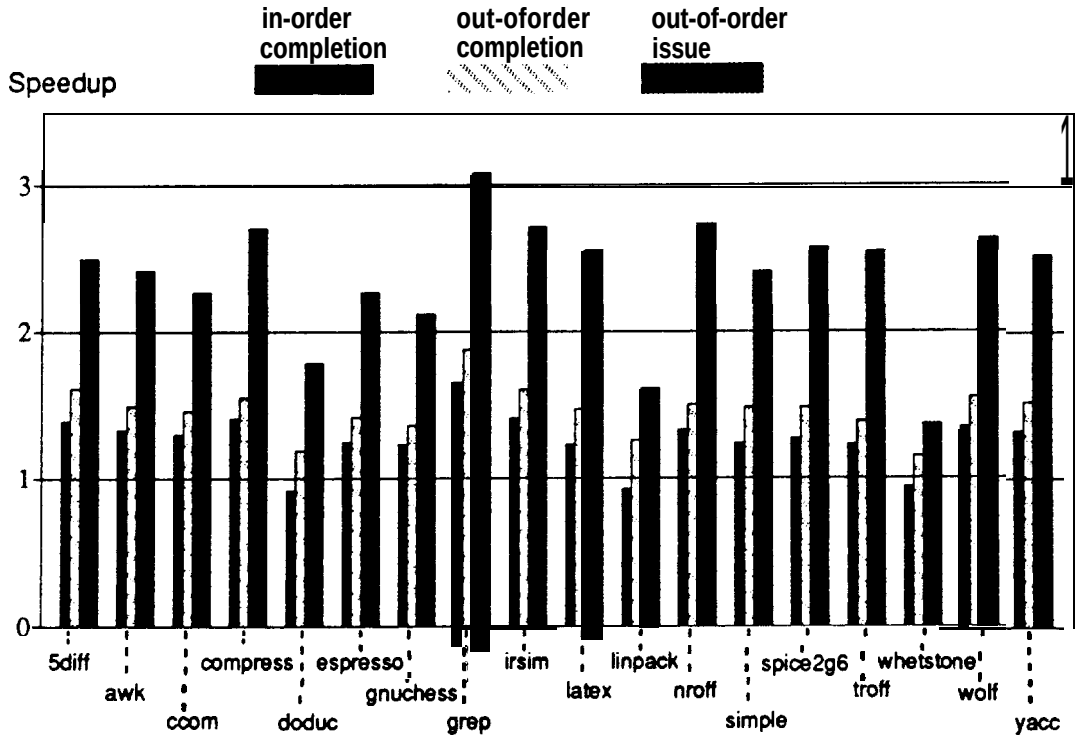

Figure 8. Potential Speedup of Three Scheduling Policies, using Ideal Instruction Fetcher

than a super-scalar processor using in-order completion. Still, out-of-order issue consistently has the best performance for the benchmark programs.

However, Figure 9 shows that out-of-order issue has much less advantage when the instruction fetcher of the super-scalar processor is modeled after the instruction fetcher of the scalar processor (the scalar fetcher, on decoding a branch instruction, waits until the execution of the branch before fetching the next sequence of instructions). Figure 9 summarizes the performance of the benchmarks with an ideal fetcher and a scalar fetcher, for all three scheduling policies. Figure 9 uses the convention—followed throughout the remainder of this thesis-of showing the low, harmonic mean, and high speedups of all benchmarks. The average speedup with out-of-order issue and an ideal fetcher is 2.4; the speedup decreases to 1.5 with a scalar fetcher. For this class of applications and a conventional instruction fetcher, the penalties incurred for procedural dependencies reduce the processor's ability to have sufficient instructions to schedule concurrently. Unless these penalties can be reduced, out-of-order issue is not cost-effective.

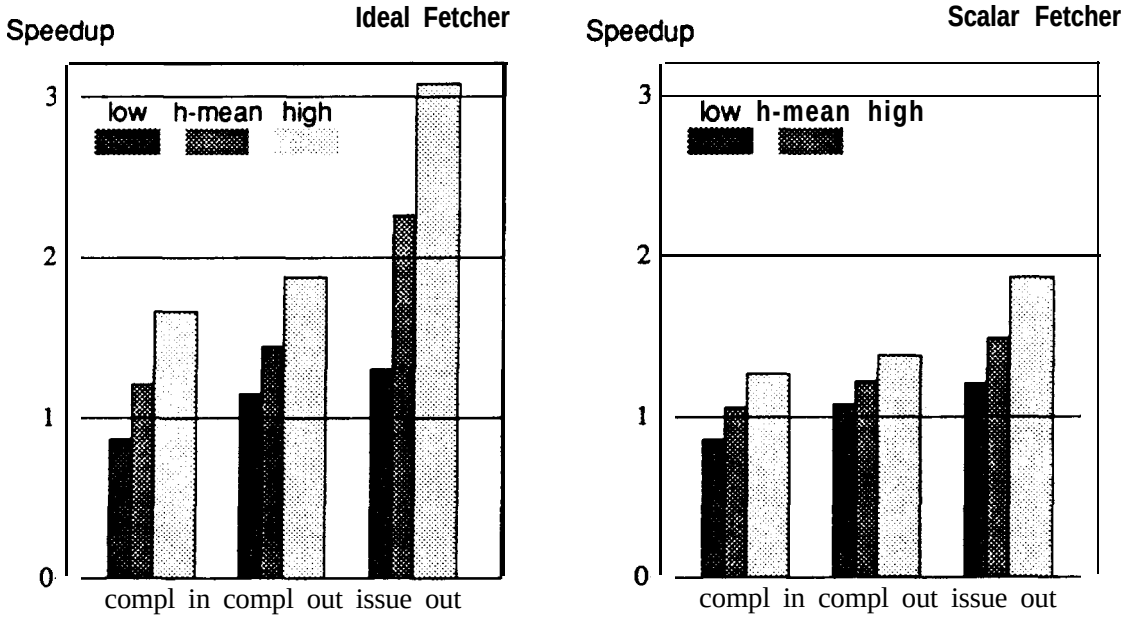

Figure 9. Speedups with Ideal Instruction Fetcher and with Instruction Fetcher Modeled after Scalar Fetcher

Chapter 4
Instruction Fetching and Decoding

Hardware scheduling is effective only when instructions can be supplied at a sufficient rate to keep the execution unit busy. If the average rate of instruction fetching is less than the average rate of instruction execution, performance is limited by instruction fetch. It is easy to provide the required instruction bandwidth for sequential instructions, because the fetcher can simply fetch several instructions per cycle. It is much more difficult to provide instruction bandwidth in the presence of non-sequential fetches caused by branches.

This chapter describes the problems related to instruction fetching in the super-scalar processor, and proposes solutions to these problems. Branch prediction allows a high instruction-fetch rate in the presence of branches, and is key to achieving performance with out-of-order issue. Furthermore, even with branch prediction, performance is highest with a wide instruction decoder (four instructions wide), because only a wide decoder can provide sufficient instruction bandwidth for the execution unit.

4.1 Branches and Instruction-Fetch Inefficiencies

Branches reduce the ability of the processor to fetch instructions because they make instruction fetching dependent on the results of instruction execution. When the outcome of a branch is not known, the instruction fetcher is stalled or may be fetching incorrect instructions, depleting the instruction window of instructions and reducing the chances that the processor can find instructions to execute concurrently. This effect is similar to the effect of a branch on a scalar processor, except that the penalty is greater in a super-scalar processor because it is attempting to fetch and execute more than one instruction per cycle.

But branches also affect the execution rate in another way that is unique to the super-scalar processor. Branches disrupt the sequentiality of instruction addressing, causing instructions to be misaligned with respect to the instruction decoder. This misalignment in turn causes some otherwise valid fetch and decode cycles to be only partially effective in supplying the processor with instructions, because the entire width of the decoder is not occupied by valid instructions.

The sequentially-fetched instructions between branches are called a ***run,*** and the number of instructions fetched sequentially is called the ***run length.*** Figure 10 shows two instruction runs occupying four instruction-cache blocks (a four-word cache block is assumed here for

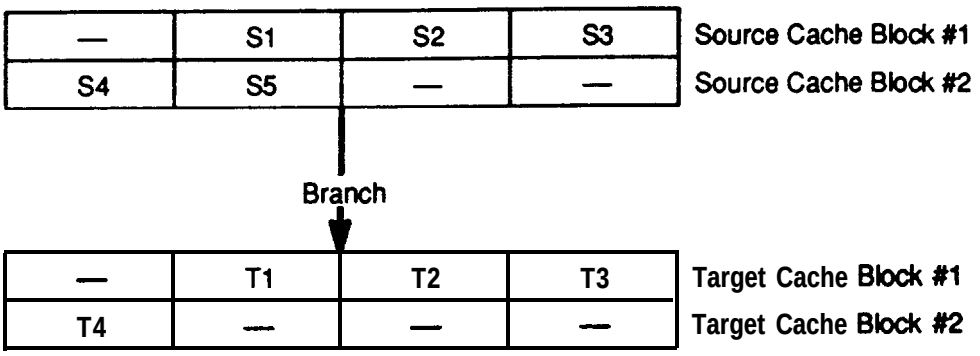

Figure IO. Sequence of Two Instruction Runs for Illustrating Decoder Behavior

the purposes of illustration). The first run consists of instructions S1–S5, which contain a branch to the second run consisting of instructions Tl-T4. Figure 11 shows how these instruction runs are sequenced through straightforward four-instruction and two-instruction decoders assuming-for illustration-that two cycles are required to determine the outcome of a branch.

The four-instruction decoder provides higher instruction bandwidth in this example-l. 125 instructions per cycle, compared to 0.9 instructions per cycle for the two-instruction decoder-but neither decoder provides adequate bandwidth to exploit the available instruction

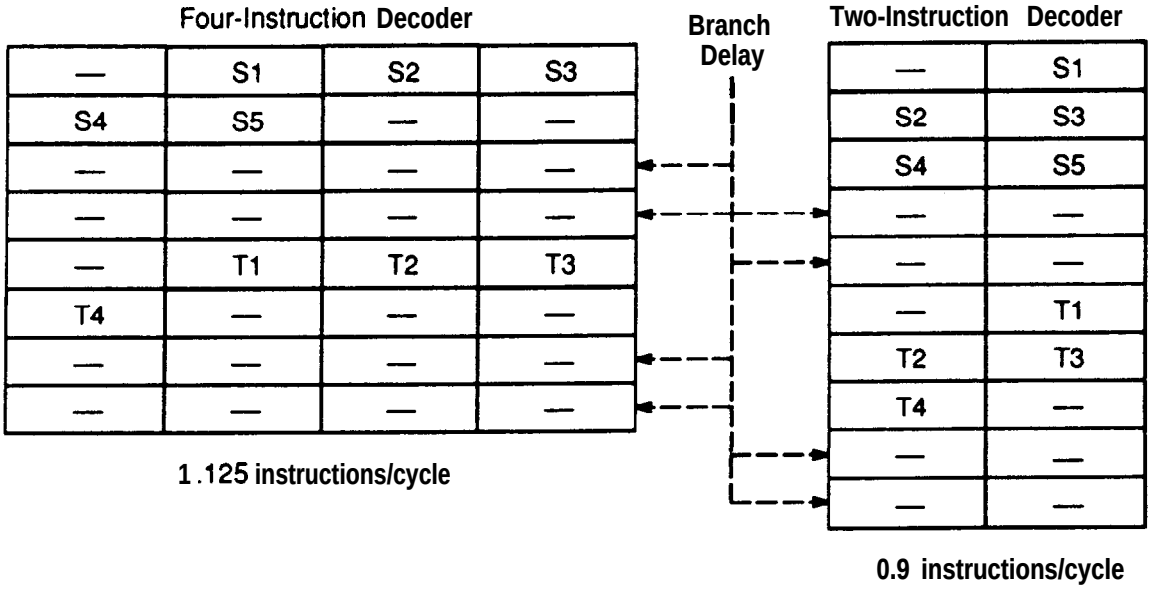

Figure I I. Sequence of Instructions in Figure 10 Through Two-Instruction and Four-Instruction Decoders

concurrency. There are two reasons for this. First, the decoder is idle while the processor determines the outcome of each branch in the sequence. Second, instruction misalignment prevents the decoder from operating at full capacity even when the decoder is processing valid instructions. For example, a four-instruction decoder spends two cycles processing the five instructions S 1–S5, even though the capacity of the decoder is eight instructions in these two cycles.

The instruction fetcher operates most efficiently when it is processing long runs of instructions. Unfortunately, general-purpose programs such as those used in this study have instruction runs that are generally quite short Figure 12 shows the dynamic run-length distributions for the sample of four typical programs (the *no-op* instructions used for pipeline scheduling in the original code are not counted in the run length). The instruction runs shown in Figure 11 are typical for the benchmark applications. In this distribution, half of the instruction runs consist of four instructions or less. The high number of short instruction runs places many demands on the instruction fetcher, if adequate fetch efficiency is to be achieved.

It is important to emphasize that the run length in Figure 12 is determined only by taken branches. In contrast, the length of basic blocks used in compiler optimizations and scheduling are determined by all branches, taken or not. The basic blocks are shorter than the

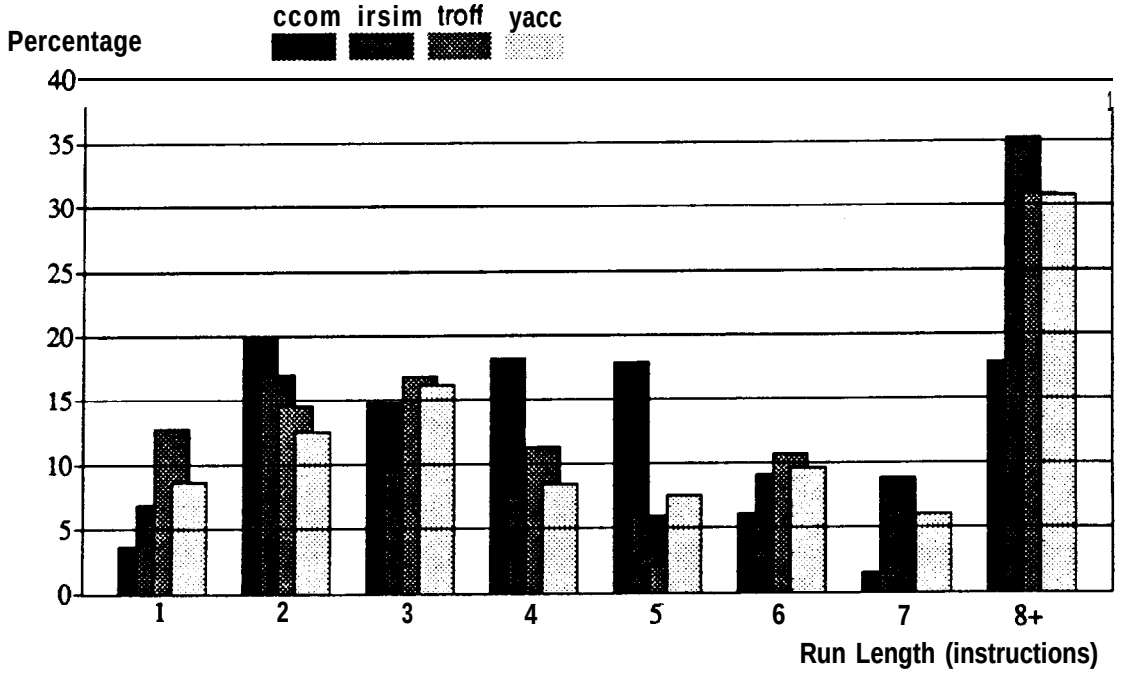

Figure 12. Dynamic Run Length Distribution for Taken Branches

instruction runs shown in Figure 12. This suggests that software would have difficulty finding sufficient instructions within a basic block to schedule effectively.

4.2 Improving Fetch Efficiency

This section discusses mechanisms which reduce the impact of branch delays and misalignment on the instruction-fetch efficiency of the super-scalar processor. It also presents the benefit of these mechanisms on processor performance. Subsequent sections consider hardware implementations of these mechanisms.

4.2.1 Scheduling Delayed Branches

A common technique for dealing with branch delays in a scalar RISC processor is to schedule branch instructions prior to the actual change of program ***flow,*** using ***delayed branches*** [Gross and Hennessy 1982]. When a delayed branch is executed, the processor initiates the fetch of the target instruction stream, but continues execution in the current stream for a number of instructions. Hence, the processor can perform useful computation during the branch delay. The number of instructions executed after the branch, but before the target, is architecture-dependent: the number is chosen according to the number of cycles required to fetch a target instruction.

A software scheduler takes advantage of delayed branches by placing useful instructions after the delayed branch instruction. These instructions are taken either from the basic block that contains the branch, or from one or both succeeding basic blocks. Scheduling more than one branch-delay instruction is most effective if the software scheduler is able to follow the branch with instructions from the likely successor block (the likely successor is determined by software branch prediction) [McFarling and Hennessy 1986]. These instructions should not be executed if the outcome of the branch is not as predicted, so the processor *squashes*, or suppresses the execution of instructions following the branch if the branch outcome is different than the outcome predicted by software. Essentially, this method of scheduling branch-delay instructions allows the processor to fetch target instructions without changing the program counter and establishing a new instruction fetch stream.

The difficulty in scheduling delayed branches to overcome the branch-delay penalty in a super-scalar processor is that the branch delay of a super-scalar processor is much higher than the delay in a scalar processor. The branch delay of a super-scalar processor is determined largely by the amount of time required to determine the outcome of the branch. In contrast, the branch delay of a scalar processor is determined largely by the time taken to fetch the

target instruction. In a super-scalar processor with out-of-order execution, the number of cycles of branch delay can be quite high. Also, because the super-scalar processor fetches more than one instruction per cycle, a very large number of instructions must be scheduled after the branch to overcome the penalty of the branch &lay.

Figure 13 plots extremes in the average number of branch-delay cycles (that is, the average number of cycles between the time that a predicted branch instruction is decoded and the time that the outcome is determined) as a function of speedup. Figure 13 also shows the extremes of instruction-fetch bandwidth sustained during this time interval. The range of speedups was obtained by varying machine configurations and instruction-fetch mechanisms over all of the sample benchmarks.

Figure 13 illustrates that the branch delay increases significantly as processor speedup increases. The increase of the branch delay with increasing speedup is the result of the buffering provided by the instruction window. With larger speedups, the instruction window is kept relatively fuller than with smaller speedups. A full window, and the dependencies a full window implies, prevent the resolution of branches for several cycles. Furthermore, during these cycles, the processor must sustain an instruction fetch rate of more than one instruction per cycle. Thus, for example, to sustain a speedup of two, software would have to schedule about eight instructions following a branch (four cycles of delay at two instructions per

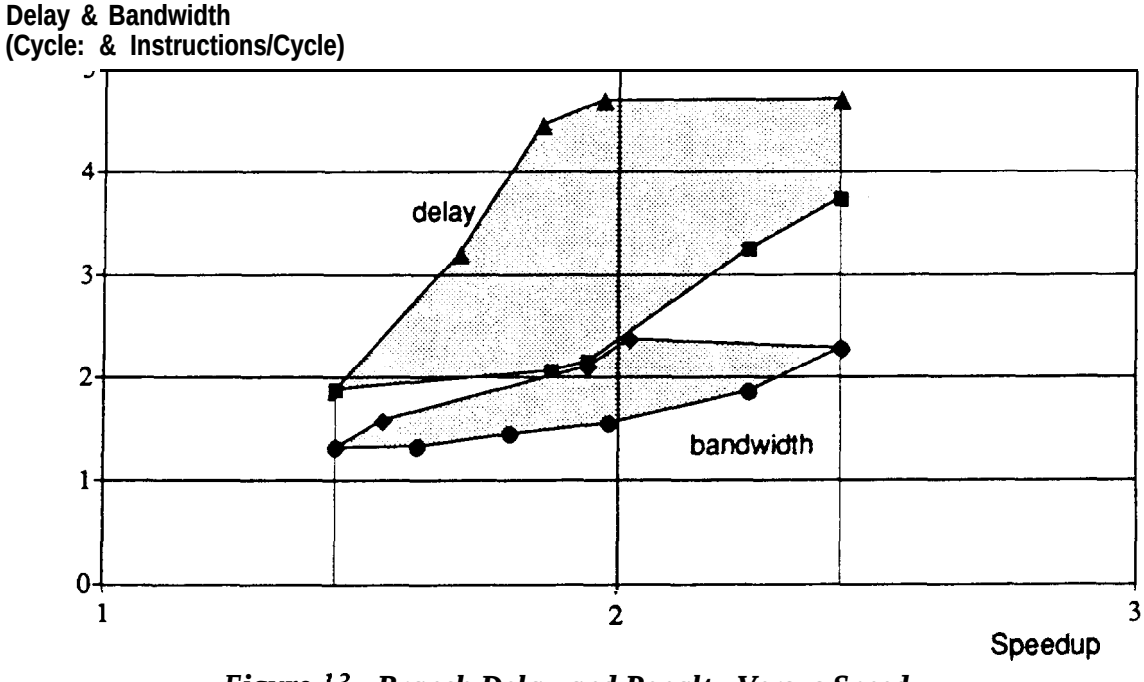

Figure 13. Branch Delay and Penalty Versus Speedy,

cycle): Pleszkun et al. [1987] demonstrate the difficulties in scheduling such a high number of instructions after a branch.

4.2.2 Branch Prediction

The instruction-fetch delays caused by branches can be reduced by predicting the outcome of the branch during instruction fetch, without waiting for the execution unit to indicate whether or not the branch should be taken. Figure 14 illustrates how this helps for the instruction sequence of Figure 11. Branch prediction relies on the fact that the future outcome of a branch can usually be predicted using knowledge of previous branch outcomes. ***Hardware branch prediction*** predicts branches dynamically using a structure such as a branch target buffer [Lee and Smith 1984], and relies on the fact that the outcome of a branch is stable over time. ***Software branchprediction*** predicts branches statically using an execution profile (or reasonable guess based on program context) to annotate the program with prediction information, and relies on the fact the outcome of a branch is usually the same.

Though they rely on different branch properties, software and hardware branch prediction yield comparable prediction accuracies [McFarling and Hennessy 1986, Ditzel and McLellan 1987]. It is important, though, that the branch-prediction algorithm not incur any delay cycles to determine the predicted branch outcome; Figure 15 illustrates the effect on fetch efficiency if a cycle is required to determine the predicted branch outcome. Any unused cycles between instruction runs cause a large reduction in fetch efficiency, because instruction runs are generally short. Hardware branch prediction has some advantage over software prediction in that it easily avoids this additional cycle. Software prediction requires an additional cycle for decoding the prediction information and computing the target address, unless the instruction format explicitly describes branches predicted to be taken (for example,

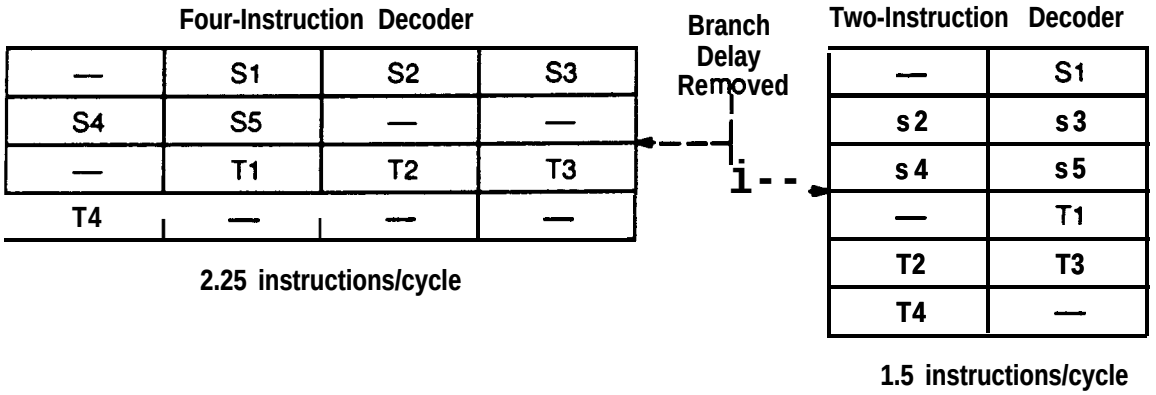

Figure 14. Sequence of Instructions in Figure II Through Two-Instruction and Four-Instruction Decoders with Branch Prediction

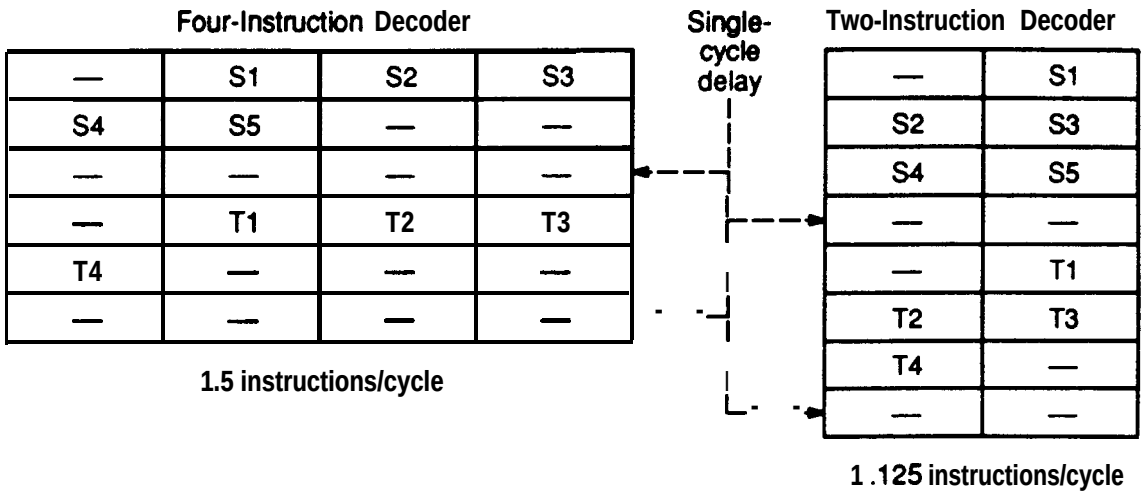

Figure 15. Sequence of Instructions in Figure 11 with Single-Cycle Delay for Decoding Branch Prediction

by a single instruction bit) and their absolute target addresses (for example, by a fixed target-address field).

Branch delays are more effectively overcome by instruction-fetch hardware that uses branch prediction than by software scheduling of branch-delay slots. The branch-prediction information can be supplied either by hardware or software, but the instruction fetcher should be designed to fetch instruction runs without any intervening delay cycles. Furthermore, the fetcher should be prepared to fetch several runs ahead of the execution unit before determining the outcome of a given branch. For example, considering that the median run is four instructions long, a four-instruction fetcher might fetch four complete instruction runs to obtain a speedup of two, because the branch delay can be more than four cycles.

Because the scope of this study does not permit the extensive evaluation of software techniques, hardware branch prediction is used. The results obtained with software prediction should not be very much different, on the average. Section 4.3 describes the branch prediction algorithm used in this study.

4.2.3 Aligning and Merging

In addition to improving the instruction-fetch efficiency by predicting branches, the instruction fetcher can improve efficiency by aligning instructions for the decoder, so that there are fewer wasted decoder slots. In the absence of branch-delay penalties, the ratio of wasted to used decoder slots depends on the particular alignment of an instruction run in memory (and, therefore, in the instruction cache). Figure 16 demonstrates the effect of instruction

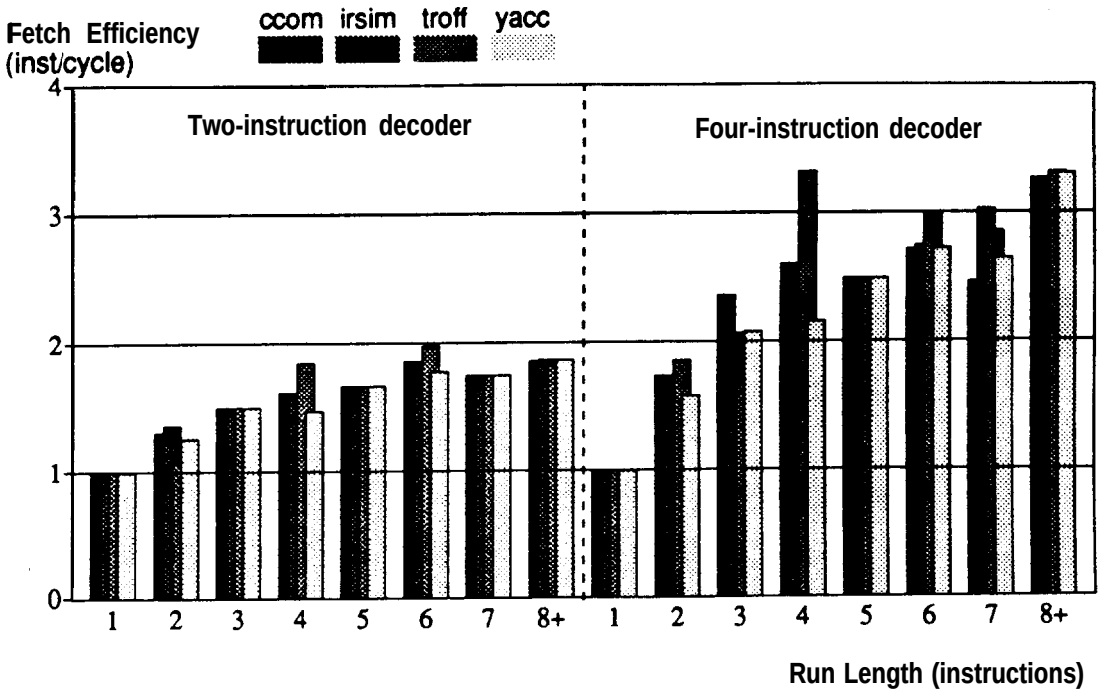

Figure 16. Fetch Efficiencies for Various Run Lengths

alignment on fetch efficiency, by showing the fetch efficiency by run length experienced during the execution of the sample benchmarks. These fetch efficiencies were determined dynamically, excluding all branch delay cycles; only misalignment causes the fetch **effi**ciencies to be less than two or four instructions per cycle. For these programs, the average fetch efficiency is 1.72 instructions per cycle for a two-instruction decoder and 2.75 instructions per cycle for a four-instruction decoder.

If the fetcher can fetch instructions faster than they are executed (for example, by fetching an entire four-word cache block in a single cycle to supply a two-instruction decoder), it can align fetched instructions to avoid wasted decoder slots. Figure 17 shows how the instruction runs of Figure 10 are aligned in a two-instruction and a four-instruction decoder (successful branch prediction is assumed for this illustration). Note that aligning has reduced the wasted decoder slots for instructions Tl-T4, but has not reduced the wasteddecoder slots for **S1–S5**.

If the fetcher performs branch prediction, it can also increase the fetch efficiency by merging instructions from different instruction runs. This is illustrated by Figure 18 (the decode slots labeled "next" in Figure 18 refer to instructions from the run of instructions following Tl-T4). As with aligning, merging depends on the ability of the fetcher to fetch instructions

Four-Instruction Decoder

S1	s2	s3	S4
s5	—	—	—
T1	T2	T3	T4

3 instructions/cycle

Two-Instnxtion Decoder

S1	S2
S3	S4
S5	—
T1	T2
T3	T4

1.8 instructions/cycle

Figure 17. Sequence of Instructions in Figure 10 Through Two-Instruction and Four-Instruction Decoders with Branch Prediction and Aligning

Four-Instruction Decoder

S1	S2	S3	S4
S5	T1	T2	T3
T4	next	next	next

4 instructions/cycle

Two-Instnrction Decoder

S1	S2
S3	S4
S5	T1
T2	T3
T4	next

2 instructions/cycle

Figure 18. Sequence of Instructions in Figure 10 Through Two-Instruction and Four-Instruction Decoders with Branch Prediction, Aligning, and Merging

at a rate greater than their execution rate, so that additional fetch cycles are available when the fetcher reaches the end of an instruction run.

4.2.4 Simulation Results and Observations

The various fetch alternatives described in Sections 4.2.2 and 4.2.3 were evaluated using the benchmark programs. These results are shown in Figure 19 and Figure 20, which plot the low, harmonic mean, and high speedups among all programs for the fetch alternatives, with both a two-instruction and four-instruction decoder. The interpretation of the chart labels is as follows:

- base — no prediction and no aligning.
- pred — predicting branches with hardware.

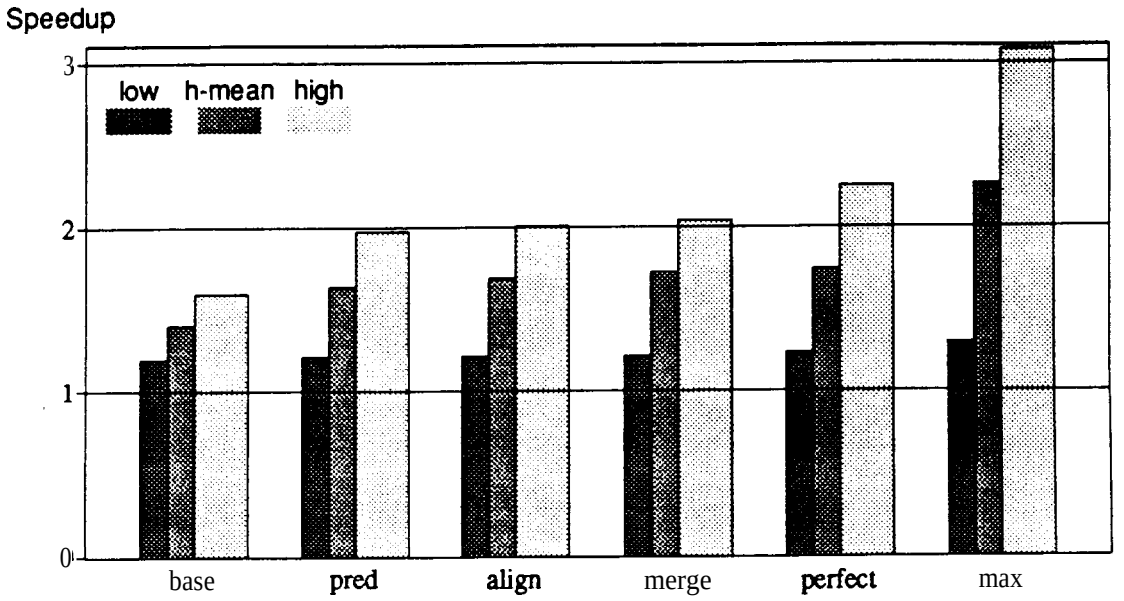

Figure 19. Speedups of Fetch Alternatives with Two-Instruction Decoder

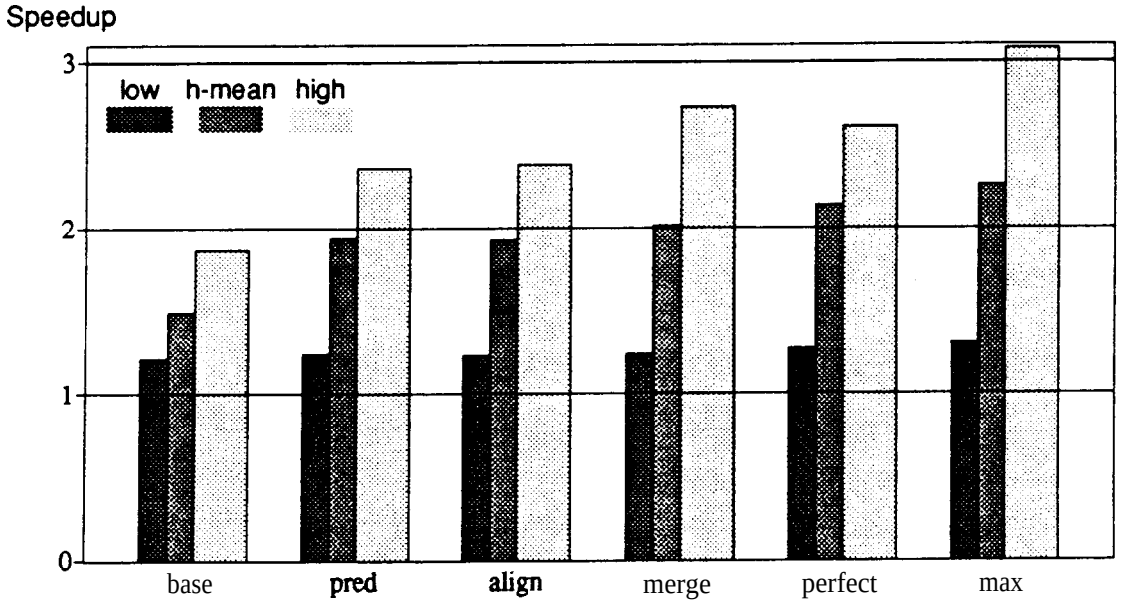

Figure 20. Speedups of Fetch Alternatives with Four-Instruction Decoder

- align — aligning the beginning of instruction runs in the decoder when possible.
- merge — merging instructions from different runs when possible.
- perfect — perfect branch prediction, but with alignment penalties. This case is included for comparison.
- [illegible] the maximum **speedup** allowed by the execution hardware, with no fetch limitations. This case is included for comparison.

The speedups for **all** but the final two sets of results are cumulative: the speedups for a given case includes the benefit of all previous cases. Aligning and merging are performed only when the **fetcher** has sufficient cycles to perform these operations.

-l-he principle observation **from** these results is that branch prediction yields the greatest incremental benefit of any of the mechanisms for improving fetch efficiency. Section 4.3 discusses a hardware implementation of branch prediction that incurs a small relative hardware cost.

Figure 19 and Figure 20 also show that a four-instruction decoder always out-performs a two-instruction decoder. This is not surprising, because the four-instruction decoder has twice the potential instruction bandwidth of the two-instruction decoder. None of the techniques applied to a two-instruction decoder to overcome this **limitation—such** as aligning and merging-yield the same advantage as a four-instruction decoder. The essential problem with a two-instruction decoder is that the instruction throughput can never exceed two instructions per cycle-the fetch efficiency is always below this limit. However, a two-instruction decoder places fewer demands on hardware than a four-instruction decoder, particularly for read ports on the register file. Section 4.4 considers the implementation of a four-instruction decoder that has reduced hardware demands.

There is some advantage in aligning and merging instruction runs, but these do not seem to be appropriate functions for hardware, because they increase the length of the critical delay path between the instruction cache and the decoder. A more efficient ***method*** to align and merge instruction runs relies on software. Software can align instructions by unrolling loops and by arranging instructions on appropriate memory boundaries. Furthermore software can merge instruction runs using software branch-prediction. If the software scheduler can effectively predict the outcome of a branch, it can move instructions from the likely successor basic block to pad the remainder of the decode slots following the branch (these instruction are decoded at the same time as the branch). Note that this is similar to scheduling

branch &lays in a scalar processor, but the goal is to improve the effkiency of the decoder. The padded instructions are executed if the prediction is correct, and are squashed if the prediction is not correct. The reorder buffer (or similar structure) in the execution unit can easily nullify a variable number of instructions following a branch, depending on the alignment and outcome of the branch.

Software aligning and merging has the advantage that instruction runs are always aligned, and are merged when a branch is successfully predicted- This is in contrast to hardware, which can perform these functions only with reserve fetch bandwidth and alignment hardware in the critical delay path between the instruction cache and decoder.

4.3 Implementing Hardware Branch Prediction

A conventional method for hardware branch-prediction uses a branch target buffer [Lee and Smith 1984] to collect information about the most-recently executed branches. Typically, the branch target buffer is accessed using an instruction address, and indicates whether or not the instruction at that address is a branch instruction. If the instruction is a branch, the branch target buffer indicates the predicted outcome and the target address. Figure 21 shows the average hits rates of a branch target buffer on the sample benchmark programs. Figure 21 is included only as an illustration: the indicated branch-prediction effectiveness

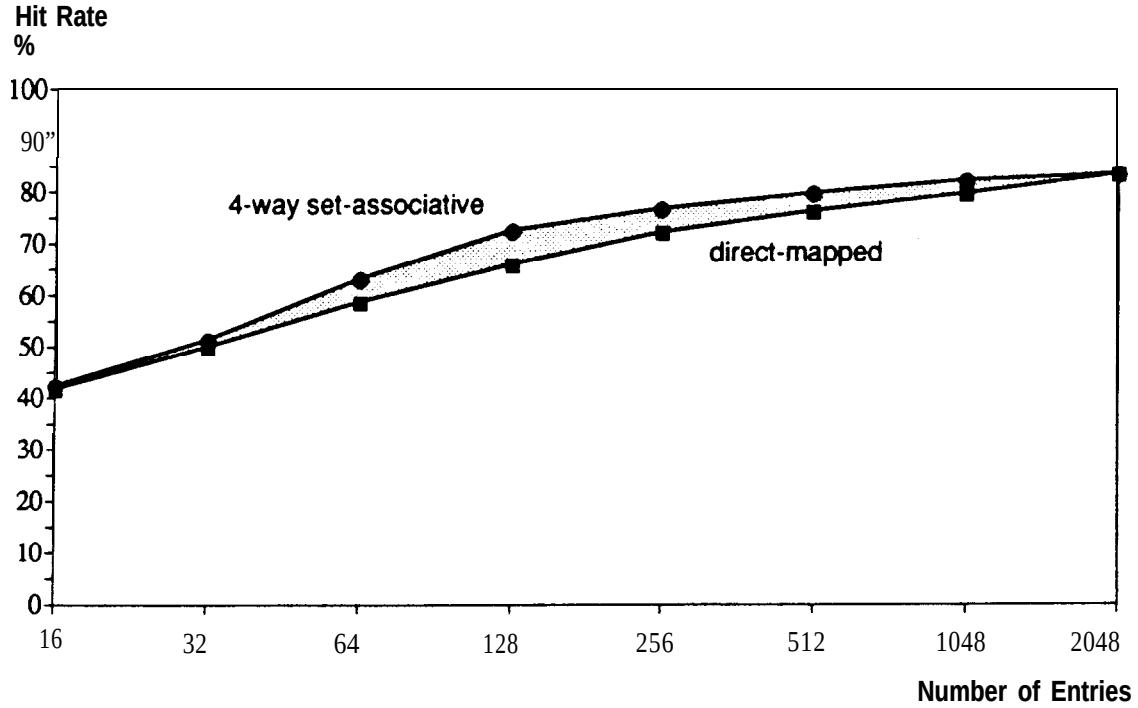

Figure 21. Average Branch Target Buffer Hit Rates

agrees with the results reported by others [Lee and Smith 1984, McFarling and Hennessy 1986, Ditzel and McLellan 1987]. A large branch target buffer is required to achieve a good hit ratio, because, unlike a cache, there is one entry per address tag and the branch target buffer cannot take advantage of spatial locality. However, the hit ratio of even a large branch target buffer is still rather low, due to misprediction.

A better hardware alternative to a branch target buffer is to include branch-prediction information in the instruction cache. Each instruction cache block includes additional fields that indicate the address of the block's successor and information indicating the location of a branch in the block. When the instruction fetch obtains a cache block containing correct information, it can easily fetch the next cache block without waiting on the decoder or execution unit to indicate the proper fetch action to be taken. This is comparable to a branch 'target buffer with a number of entries equal to the number of blocks in the instruction cache and with associativity equal to the associativity of the instruction cache. This scheme is nearly identical to the branch target buffer in prediction accuracy (most benchmark pro-g-rams experienced a branch prediction accuracy of 80–95% in this study).

Indirect branches (that is, branches whose target addresses are specified by the contents of registers) can reduce the branch prediction ratio even when they are unconditional, because branch-prediction hardware cannot always predict the successor cache block when the target address can change during execution. However, the target addresses of most dynamic indirect branches show locality because of repetitive procedure calls from the same procedure, making it useful to predict such branches if the cost is not high (the benefit is about a 2-3% improvement in branch-prediction accuracy).

The results given in Section 4.2.4 assume the existence of an instruction fetcher that can fetch instructions without any delay for successfully-predicted branches. 'l-his section discusses the implementation of this instruction fetcher to demonstrate that an implementation is feasible and to show that it causes only a small increase in the size of the instruction cache. Also, this section shows that the fetcher can predict indirect branches with no additional cost over that required for the basic implementation.

4.3.1 Basic Organization

Figure 22 shows a sample organization for the instruction-cache entry required by the instruction fetcher (this entry may include other information not important for this discussion). For this example, the cache entry holds four instructions. The entry also contains instruction-fetch information which is shown expanded in Figure 22. The fetch information

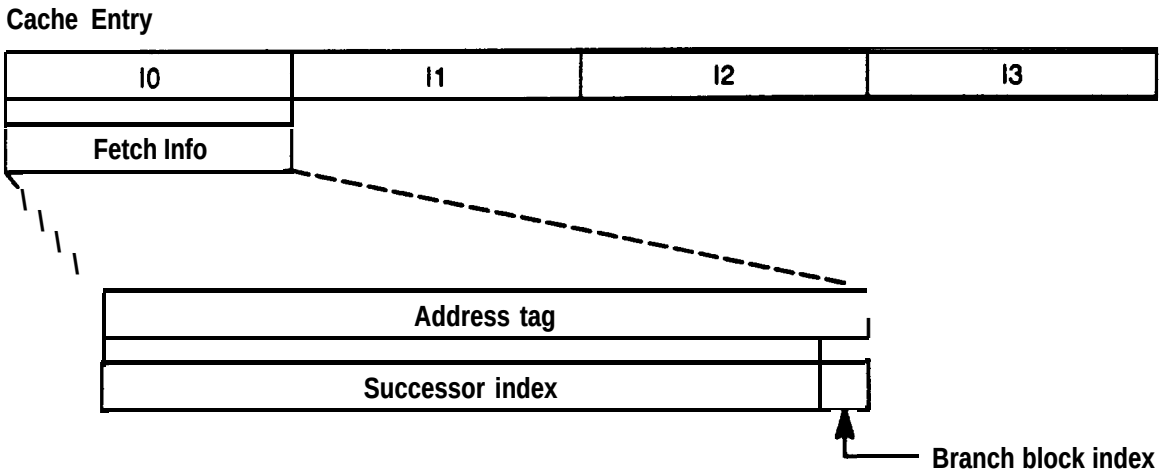

Figure 22. Instruction Cache Entry for Branch Prediction

contains an address tag (used in the normal fashion) and two additional fields used by the instruction fetcher:

- The ***successor index*** field indicates both the next entry predicted to be fetched and the first instruction within this next entry predicted to be executed. The successor index does not specify a full instruction address, but is of sufficient size to select any instruction within the cache. For example, a 64 Kbyte, direct-mapped cache requires a 14-bit successor index if all instructions are 32 bits in length.

- ***The branch block index*** field indicates the location of a branch point within the instruction block. Instructions beyond the branch point are predicted not to be executed- Figure 23 shows sample instruction-cache entries for the code sequence of Figure 14, assuming a 64Kbyte, direct-mapped cache and with the indicated instruction addresses.

4.3.2 Setting and Interpreting Cache Entries

When a cache entry is first loaded, the cache fetch hardware sets the address tag in the normal manner, and sets the successor index field to the next sequential fetch address. The default for a newly-loaded entry, therefore, is to predict that branches are not taken. If the prediction is incorrect, this will be discovered later by the normal procedure for detecting a mispredicted branch.

As Figure 23 illustrates, a target program counter can be constructed at branch points by concatenating the successor index field of the branching entry to the address tag of the

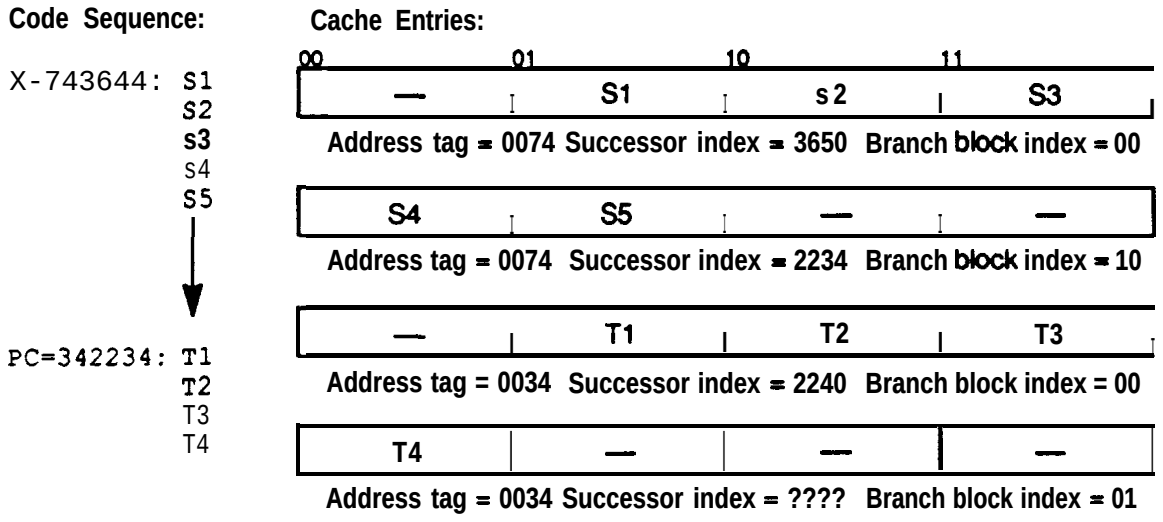

Figure 23. Example Cache Entriesfor Code Sequence of Figure 14

successor entry. Between branch points, the program counter is incremented and used to detect cache misses for sequential runs of instructions. Cache misses for branch targets are handled as part of branch prediction checking. The program counter recovered from the cache entries can be used during instruction decode to compute program-counter-relative addresses and procedure return addresses; if the program counter is wrong because of misprediction, this will be detected later. Note that, for a set-associative cache, some bits in the successor index field are used to select a block within a cache set, and are not involved in generating the program counter.

The validity of instructions at the beginning of a cache entry are determined by low-order bits of the successor index field in the preceding entry. When the preceding entry predicted a taken branch, this entry's successor index may point to any instruction within the current block, and instructions up to this point in the block are not to be executed- The validity of instructions at the end of the block are determined by the branch block index, which indicate the point where a branch is predicted to be taken (the value 00 is used when that there is no branch within the block). The branch block index is required by the instruction decoder to determine valid instructions, and is not used by the instruction fetcher. The instruction fetcher retrieves cache entries based on the successor in&x fields alone.

4.3.3 Predicting Branches

To check branch predictions, the processor keeps a list of predicted branches ordered by the sequence in which branches are predicted. Each entry on this list indicates the location of the

branch in the cache; this location is identified by concatenating the successor index of the entry preceding the branching entry with the branch block index. Each entry also contains a complete program-counter value for the target of the branch. Note that because the cache only predicts taken branches this list only contains taken branches.

The processor executes all branches in their original program sequence (this is guaranteed by the operation of the branch reservation station). Note that these branches are detected by instruction decoding, independent of prediction information maintained by the instruction fetcher. When a branch is executed, the processor compares information related to this branch with the information at the head of the list of predicted branches. The following conditions must hold for a successful prediction:

- If the branch is taken, its location in the cache must match the location of the next branch on the list of predictions. -l-his is required to detect a taken branch that was predicted to be not taken.
- The predicted target address of the branch at the head of the list must match the next instruction address determined by executing the branch. This comparison is relevant only if the location of the branch matches the location of the next branch on the list of predictions, and is required primarily to detect **a** non-taken branch that was predicted to be taken. However, since the predicted target address is based on the address tag of the successor block, this comparison also detects that cache replacement has removed the original target entry. In addition, comparing target addresses checks that indirect branches were properly predicted.

If either or both of the above conditions does not hold, the instruction fetcher has mispredicted a branch. The instruction fetcher uses the location of the branch determined by the execution unit to update the appropriate cache entry.

4.3.4 Hardware and Performance Costs

The principal hardware cost of the proposed branch-prediction scheme is the increase in the cache size caused by the successor index and branch block index fields in each entry. For a 64 Kbyte, direct-mapped cache, these add about 11% to the cache storage required. However, the performance increase obtained by branch prediction seems worth this cost, especially considering the normal size/performance tradeoffs involved in cache design [Przybylski et al. 1988].

This scheme retains branch history only to the extent of tracking the most recent taken branches. Branch-prediction accuracy can be increased by retaining more branch-history

information [Lee and Smith 1984], but this is significantly more expensive in the current proposal than in most other proposals. The current proposal saves storage by closely associating branch-prediction information with a cache block: it predicts only one taken branch per cache block, and predicts non-taken branches by not storing any branch information with the block. This has the advantage that any number of non-taken branches can be predicted without contending with taken branches for cache entries. Figure 24 illustrates, with the set of bars labeled "predict all," the reduction in performance that would result if information were retained for every branch rather than just taken branches. The reduction in performance is due solely to contention for prediction entries between branches in the same cache block. This contention can be reduced by additional entries that provide space to hold branch information for more than one branch per block, but the slight improvement in prediction accuracy of more complex branch-prediction schemes (2-3% at best) is not worth the additional storage and complexity.

The requirement to update the cache entry when a branch is mispredicted conflicts with the requirement to fetch the correct branch target. Unless it is possible to read and write the fetch information for two different entries simultaneously, the updating of the fetch information on a mispredicted branch takes a cycle away from instruction fetching. Figure 25 indicates, with the set of bars labeled "update pred," the effect that this fetch stall has on performance

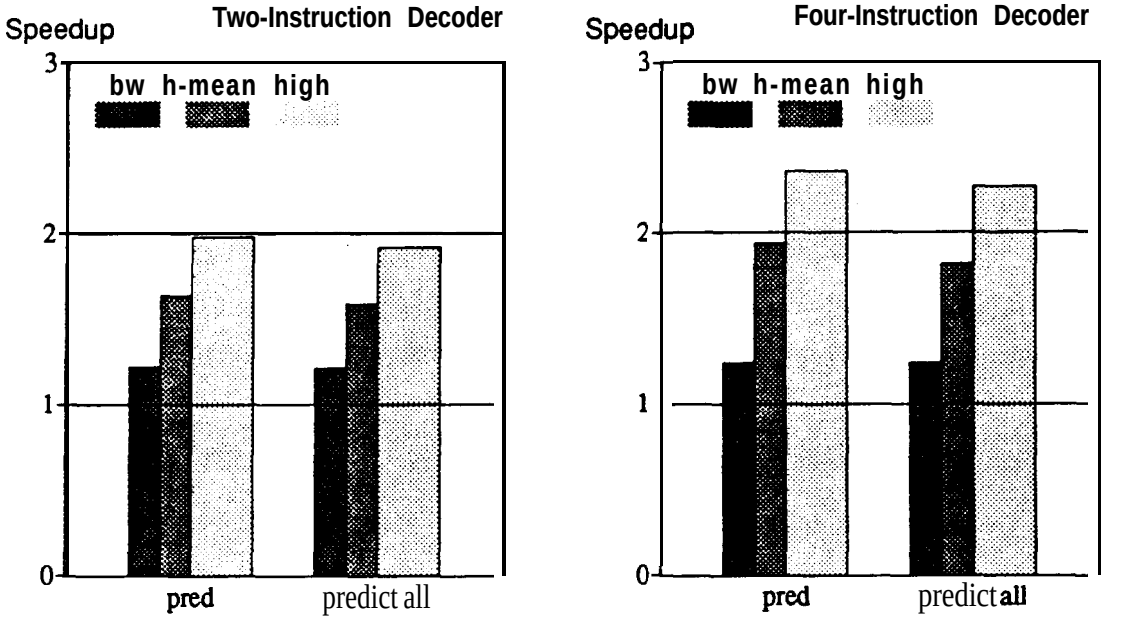

Figure 24. Performance Decrease Caused by Storing All Branch Predictions with Cache Blocks

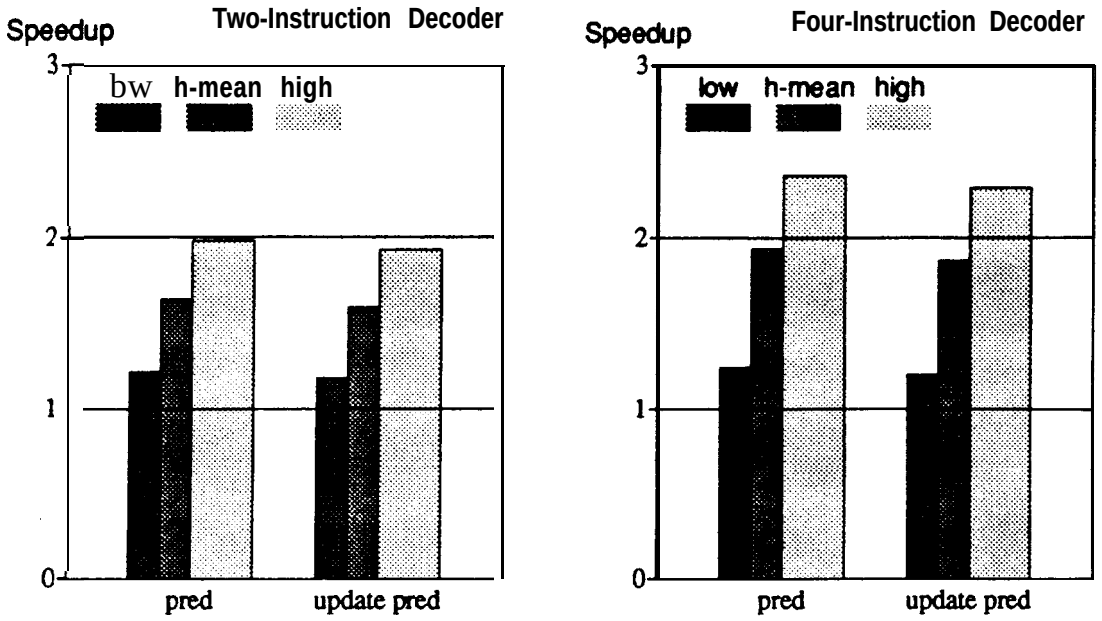

Figure 25. Performance Degradation with Single-Port Cache Tags

for a two-instruction and a four-instruction decoder. For comparison, the results of the "pred" case from Figure 20 is repeated in Figure 25. The additional fetch stall causes only a small reduction in performance: 3% with a two-instruction decoder and 4% with a four-instruction decoder. -l-his is reasonable, since the penalty for a mispredicted branch is high (see Figure 13), but is incurred infrequently; the additional cycle represents only a small proportional increase in the penalty.

4.4 Implementing a Four-Instruction Decoder

Section 4.2.4 showed that a four-instruction decoder yields higher performance (20–25% higher) than a two-instruction decoder, because it is less sensitive to instruction alignment than the two-instruction decoder. However, directly modeling a four-instruction decoder after a single-instruction decoder is not cost-effective. In a straightforward implementation, decoding four instructions per cycle requires eight read ports on both the register file and the reorder buffer, and eight buses for distributing operands. Furthermore, with this four-instruction decoder, the execution hardware described in Section 3.3 requires a total of 210 comparators for dependency analysis (192 in the result buffer and 18 to check dependencies between decoded instructions). At the same time, the performance benefit is limited by instruction fetching and dependencies. Doubling the amount of hardware over a two-

instruction decoder for a **20–25%** performance benefit as shown in Figure 20 (which includes the assumption that cycle time is not affected) hardly seems justified.

Furthermore, most of the capability of a straightforward four-instruction decoder is wasted. Figure 26 shows the demand on register-file operands caused by a four-instruction decoder during the execution of the sample benchmarks. This distribution was measured with the decode stage occupied by valid instructions on every cycle (there are no branch delay cycles, although there are alignment penalties), so this is an upper bound There are several reasons that the register demand is so low:

- not all decoded instructions access two registers,
- not all decoded instructions are valid (due to misalignment), and
- some decoded instructions have dependencies on one or more simultaneously-decoded instructions (the corresponding operands are obtained later by result forwarding).

It is possible to take advantage of the relatively low demand for register-file read ports if the decoder provides a limited number of read ports that are scheduled as other resources are. Access to the register-dependency logic is also limited in this manner, since **register-**

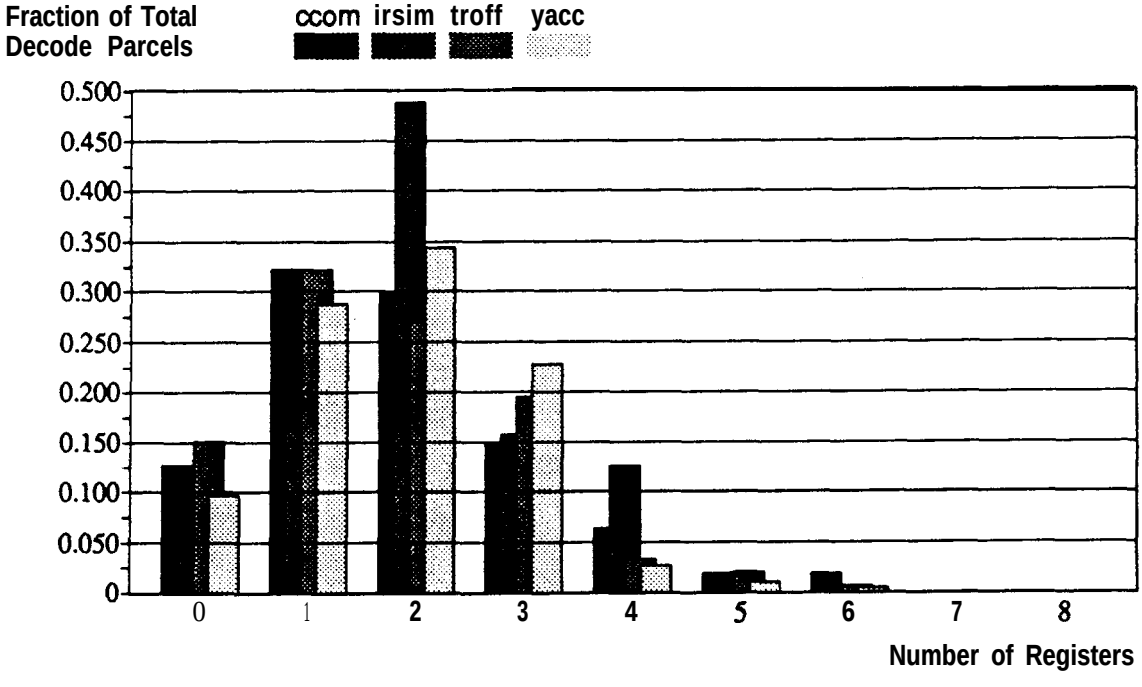

Figure 26. Register Usage Distribution of a Four-Instruction Decoder—No Branch Delays

dependency logic is closely associated with register access. Figure 27 shows the results of constraining the number of registers available to a four-instruction decoder. This constraint reduces performance by less than 2%.

As will be shown in this section, arbitration for register-file read ports is difficult if it is performed by hardware alone. However, it is conceptually easy for a software scheduler to know which instructions will be decoded simultaneously and to group instructions so that register demands are limited for each group of instructions. Software can also facilitate the detection of register dependencies between simultaneously-decoded instructions.

4.4.1 Implementing a Hardware Register-Port Arbiter

Arbitrating for register-file ports requires that the decoder implement a prioritized select of the register identifiers to be applied to the register file. The register file is typically accessed during the second half of the decode cycle, so the register identifiers must be valid at the register file by the mid-point of the decode cycle. For this reason, the prioritized register-identifier selection must be accomplished within about half of the processor cycle.

It is easy to design the instruction encoding so that register requirements are known early in the decode cycle. However, it is difficult to arbitrate register access among contending instructions. Table 4 shows an estimate of the amount of logic required to implement the

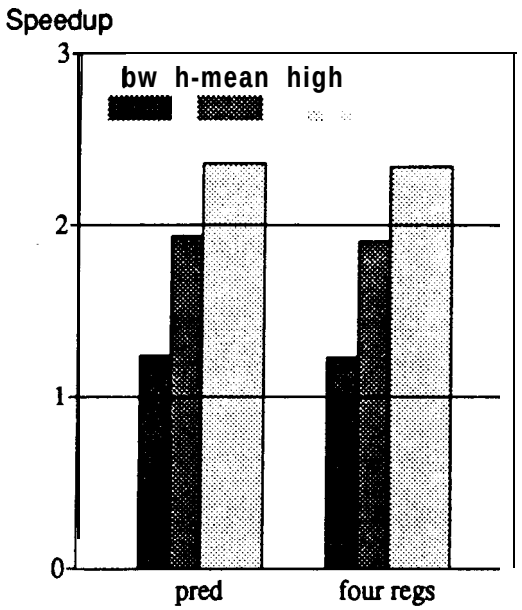

Figure 27. Performance Degradation Caused by Limiting a Four-Instruction Deco&r to Four Register-File Ports

Table 4. Estimate of Register-Port Arbiter Size: Two Logic Levels

Decode Reg #:	1	2	3	4	5	6	7	8
Enable to Port 0	—	1/1	1/2	1/3	1/4	1/5	1/6	1/7
Enable to Port 1		1/1	2/2	3/3	4/4	5/5	6/6	7/7
Enable to Port 2			1/2	3/3	6/4	10/5	15/6	21/7
Enable to Port 3				1/3	4/4	10/5	20/6	35/7
Totals:		2/1	4/2	8/3	15/4	26/5	42/6	64/7

Note: the first of each table entry is the number of gates required in the first level of logic, and the second is the number of inputs required by each gate. The size of the gates in the second logic level are indicated by the total number of gates in the first level.

prioritized register-identifier selection in two levels of logic. The eight register operands possibly required by a four-instruction decoder are shown across the top of Table 4. Each row of the Table 4 shows, for each possible register operand, the number of gates and the number of inputs of each gate required to generate an enable signal. This signal selects the given register identifier at one input of an eight-to-one multiplexer, enabling the given operand to be accessed at the indicated register-file port. This logic follows the following general form:

- if the **first** register operand requires a register access, it is always enabled on the first port.
- if the second register operand requires a register access, it is enabled on the **first** port if the first register operand does not require an access, and is otherwise enabled on the second port.
- in general, the register identifier for a required access is enabled on the first port if no other previous operand requires an access, on the second port if one previous operand requires an access, on the third port if two previous operands use ports, on the fourth port if three previous operands use ports, and on no port (and the decoder stalls) if four previous operands use ports.

This type of arbitration occurs frequently in the super-scalar processor, to resolve contention for shared resources. Resolving contention within ***two*** logic levels requires $order(mn^3)$ gates where m is the number of resources being contended for (in this case, four register-file read ports) and n is the number of contenders (in this case, eight register-operand identifiers). A factor mn results from generation of an enable per contender per resource, and a factor of n^2 results from interaction between contenders. Since the largest first-level gate

requires $n-1$ inputs, the amount of chip area taken by the first-level gates is $order(mn^4)$. The size and irregularity of the prioritizer argue for implementing it with a logic array. However, regardless of the implementation, the prioritizer is likely to be slow.

The prioritizer is much smaller and more regular if it is implemented in serial logic stages. In this implementation, each register operand receives a port identifier from the preceding, adjacent operand. The operand either uses this port and passes the identifier for the next available port (or a disable) to its successor, or does not use the port and passes the port identifier unmodified to its successor. The obvious difficulty with this approach is the number of levels required: the register-port arbiter requires approximately fourteen logic levels using this approach. Thus, even though the number of register-file ports can be reduced, the arbiter required by this approach is difficult to implement, given the timing constraints. The following section considers an alternative implementation that relies much more on software, and which greatly simplifies the implementation of the decoder.

4.4.2 Limiting Register Access Via Instruction Format

Figure 28 illustrates an instruction format that facilitates restricting the number of registers accessed by a four-instruction decoder. The instructions which occupy a single decode stage are grouped together, with a separate *register access* field. The register access field specifies the register identifiers for four source operands and the register identifiers for four destination registers. Each destination-register identifier corresponds, by position, to an instruction in the decoder, and instructions do not need to identify destination registers. As Figure 29 shows, each instruction identifies source operands by selecting among the source- and destination-register identifiers in the register access field. Identifying the destination register of a previous instruction as a source indicates a dependency on the corresponding result value.

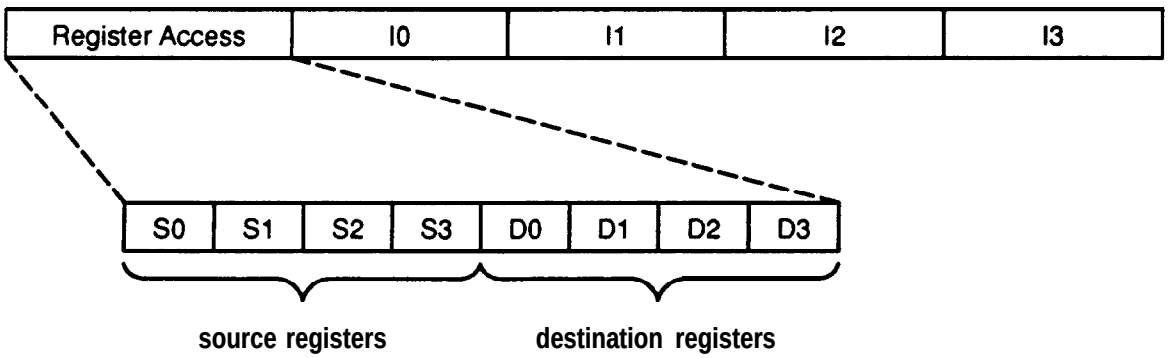

Figure 28. Format for Four-Instruction Group Limiting the Number of Registers Accessed

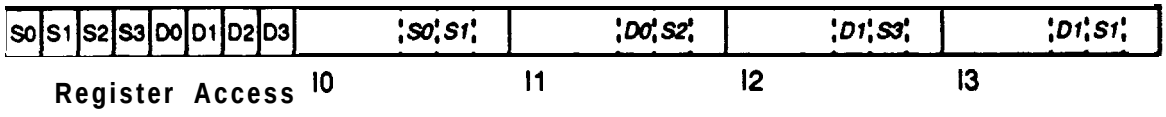

Figure 29. Example Operand Encoding using Instruction Format of Figure 28

For example, in Figure 29, the second instruction depends on the result of the first instruction, because the field ***DO*** specifies a source operand of the second instruction.

With this approach, the decoder is easily implemented. The source-register identifiers in the register access field are applied to the register file, and each instruction simply selects operands from among the accessed operands. The decoder cannot stall because too many registers are required (software has inserted ***no-ops*** for this purpose, as explained below).

This approach has one drawback compared to the hardware approach described in Section 4.4.1. It does not allow register usage to be reduced because of instruction misalignment. Software cannot know about dynamic branch activity, and must assume that all decode slots are fully occupied. However, this approach has an advantage over that of Section 4.4.1 in that it allows several instructions within the group to access the same operand register without using additional register-file ports. This sharing is difficult to accomplish in hardware, requiring 28 comparators for detecting matching source-register identifiers and complicating register-port arbitration to allow register-port sharing for common identifiers.

To take advantage of this instruction format, the compiler or a post-processor groups instructions by decoder boundaries, and arranges the register usage of these instructions so that the group of instructions accesses no more than four total register operands. Meeting this restriction may require the insertion of ***no-op*** instructions into the group. Having branches appear within the group does not cause any difficulties: whether or not the branch is taken during execution, the instructions following the branch can indicate as source operands the destination registers of instructions preceding the branch.

However, branch targets within the instruction group interfere with the goal of statically encoding instruction dependencies. If an instruction in the group is the target of a branch, the instructions preceding the target may or may not be executed, depending on the path of execution through the block. Thus, an instruction following the target cannot statically indicate a dependency on an instruction preceding the target: the dependency may or may not exist within the group, depending on the execution path. There are several possible solutions to this problem:

- Software can avoid indicating dependencies between instructions which have an intervening branch target, and simply use a separate source-register field for the dependent instruction. **In this** case, hardware **must be able to** detect whether or not a dependency exists during instruction decode. The instruction format loses the advantage of indicating dependencies, but **still** retains the advantage of reducing register-port requirements.

- Hardware can rearrange register identifiers within the instruction depending on the path of execution. For example, **destination-register identifiers of un-**executed instructions preceding a branch target can be moved to corresponding source-register fields; instructions that refer to these fields are also changed to reflect the moved identifier. **Figure** 30 shows how the hardware might change the instructions in the example of Figure 29 when the second instruction is the target of a branch. This technique relaxes the constraints on software, but does not remove the constraints. For example, as Figure 30 **illustrates,** if 12 were a branch target and the hardware simply moved destination register identifiers of IO and I1, the instruction 13 would not correctly obtain its second source operand.

- Software can avoid branch targets within an instruction group. This is consistent with the desire to have software align branch targets on decoder boundaries to improve instruction-fetch efficiency.

These alternatives trade off hardware complexity for software constraints. Given that branches **already** constrain instruction fetching in a number of ways, it is likely that a few additional constraints imposed on software do not reduce performance very much. If software can successfully align instructions avoid branch targets within instruction groups, the third option above is best. Otherwise, the second option is preferred over the first.

4.5 Implementing Branches

The techniques for performing branches in a scalar RISC processor cannot be used directly in the super-scalar processor. The super-scalar processor may decode more than one branch

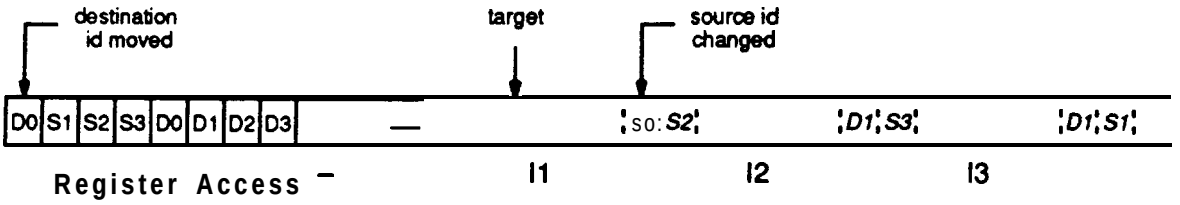

Figure 30. One Approach to Handling a Branch Target Within Instruction Group

per cycle, may have to retain a branch instruction several cycles before it is executed, and may have several unexecuted branches pending at any given time. This section considers the performance value of these complications, and ways to simplify branch decode and execution. To prevent the comparison of the super-scalar processor to the scalar processor from including unrelated effects, the branch instructions and the pipeline timing of branches are assumed to be equivalent to those of the R2000 processor (3) .

4.5.1 Number of Pending Branches

Having a number of outstanding, unexecuted branches in the processor is a natural result of using branch prediction to overcome branch delay. Between the time that a branch is decoded and its outcome is determined, one or more subsequent branches may be decoded. For best instruction throughput, subsequent branches should also be predicted and removed from the decoder. This not only avoids decoder stalls, but also provides additional instructions for scheduling in the likely event that subsequent branches are predicted correctly. However, the number of pending branches directly determines the size of the branch reservation station and the size of the branch-prediction list.

Figure 31 shows the effects on performance as the number of outstanding branches is decreased. The processor hardware used to obtain these results has six reservation-station entries for branches, rather than four, to accommodate all possible outstanding branches. With either a two-instruction or four-instruction decoder, nearly maximum performance is achieved by allowing up to four outstanding branches. In this case, the super-scalar processor is scheduling useful instructions from as many as five different instruction runs at once.

4.5.2 Order of Branch Execution

As long as branches are predicted correctly, branches can be executed in any order, and multiple branches can be executed per cycle. This improves instruction throughput and decreases the branch-resolution penalty by decreasing the chance that a mispredicted branch has to wait several cycles while previous, successfully-predicted branches complete sequentially. Of course, if any branch is mispredicted, all subsequent results must be discarded even though intervening branches appeared to be correctly predicted. (Uht [1986] describes

(3) Many of the branch instructions in the R2000 perform predicate operations as well as branching, and these branches require a separate (but simple) functional unit to perform these operations (the predicate operations cannot be performed by the ALU without conflicts).

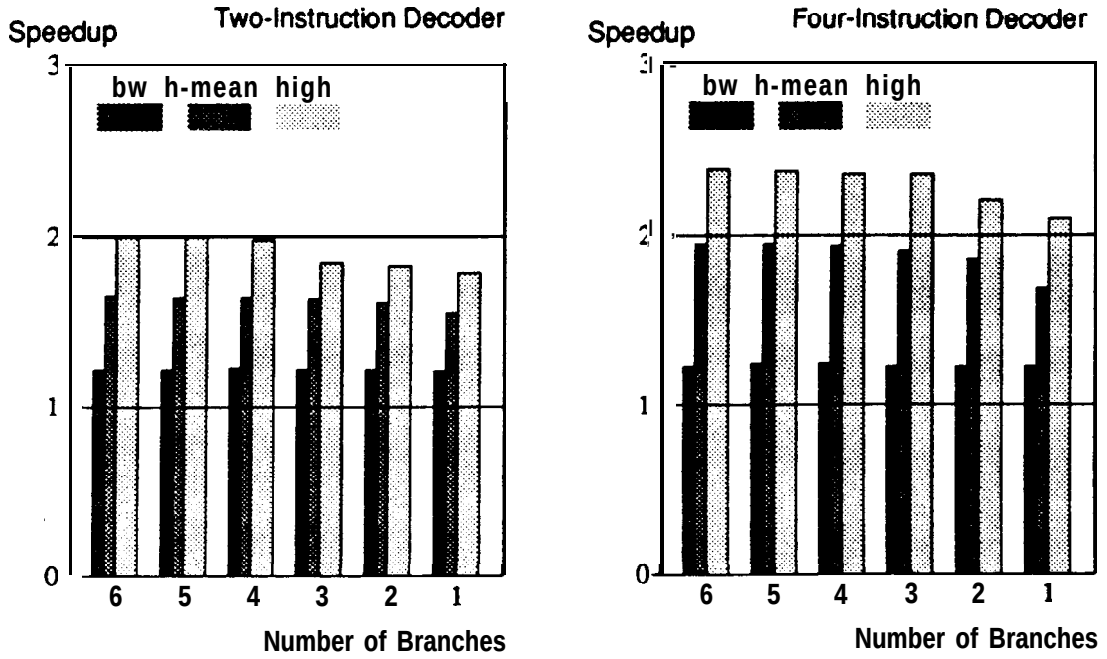

Figure 31. Reducing the Number of Outstanding Branches

a more general method for relaxing constraints on the issuing of independent branches, but this relies on keeping instructions in lexical order.)

Figure 32 shows the performance benefit of executing multiple, out-of-order branches per cycle. There is no increase in performance for a two-instruction decoder, because performance is limited by instruction fetching. And, although there is a slight increase in performance with a four-instruction decoder (about 3%), the increase is not large enough to warrant the additional hardware to schedule and execute multiple branches per cycle.

4.5.3 Simplifying Branch Decoding

Implementing a minimum branch decode-to-execution delay requires that the branch target address be determined during the decode of a branch. The target address may be needed as soon as the following cycle, for detecting a misprediction and fetching the correct target instructions. The R2000 instruction set follows the common practice of computing branch target addresses by adding program-counter-relative displacements to the current value of the program counter. Since the super-scalar processor decodes more than one branch per cycle, computing potential branch target addresses for all decoded instructions would require an adder per decoded instruction.

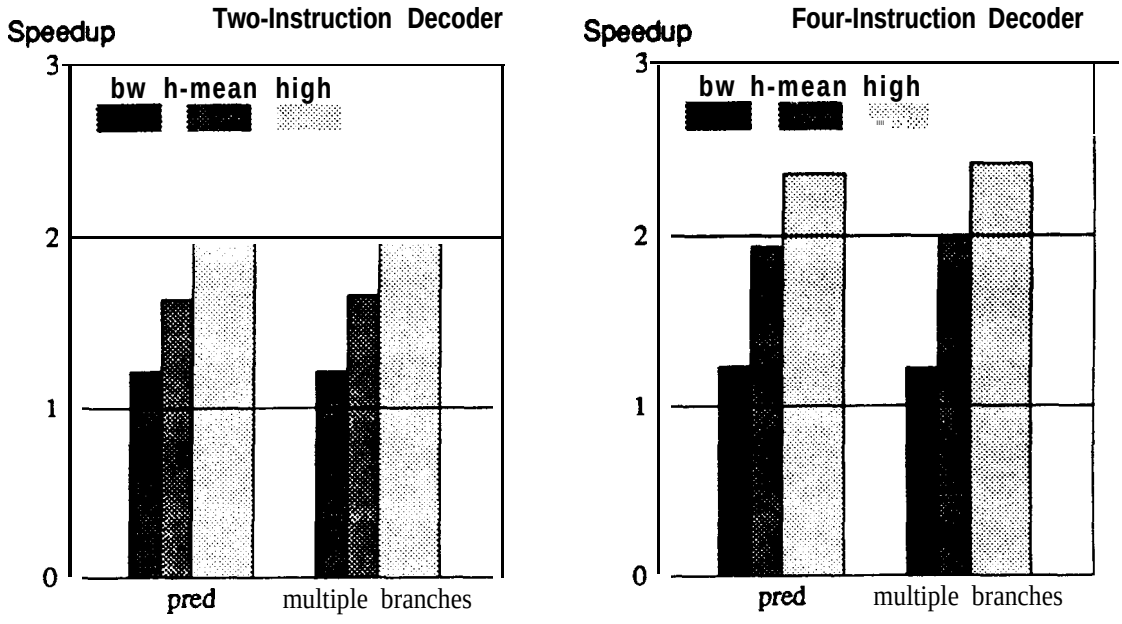

Figure 32. Performance Increase by Executing Multiple Correctly-Predicted Branches Per Cycle

Fortunately, there is not a strong need to compute more than one branch target address per decode cycle. As Figure 33 shows, there is only a slight performance decrease (about 2%) caused by imposing a limit on the decoder of one target address per cycle. The decrease is slight because the branch-prediction logic is limited to predicting one branch per cycle, and the execution hardware is limited to executing only one branch per cycle. Placing the additional limit in the decode stage may occasionally stall the decoding of a second branch and subsequent instructions, but this limit is not severe compared to the other branch limits. The limit can be avoided if the relative-address computation is performed during the execution of the branch rather than during decode, in which case all decoded branches are simply placed into the branch reservation station. This adds a cycle to the best-case branch delay, and increases the penalty for a mispredicted branch.

Even though there is only one target-address adder, computing the target address is still more complex than in the scalar processor. Instructions in the decoder must arbitrate for the address-computation hardware, with the first branch instruction having priority. Also, the program-counter-relative computation must take into account the position of the branch instruction in the decoder, because there is only a single program counter for the entire decode stage.

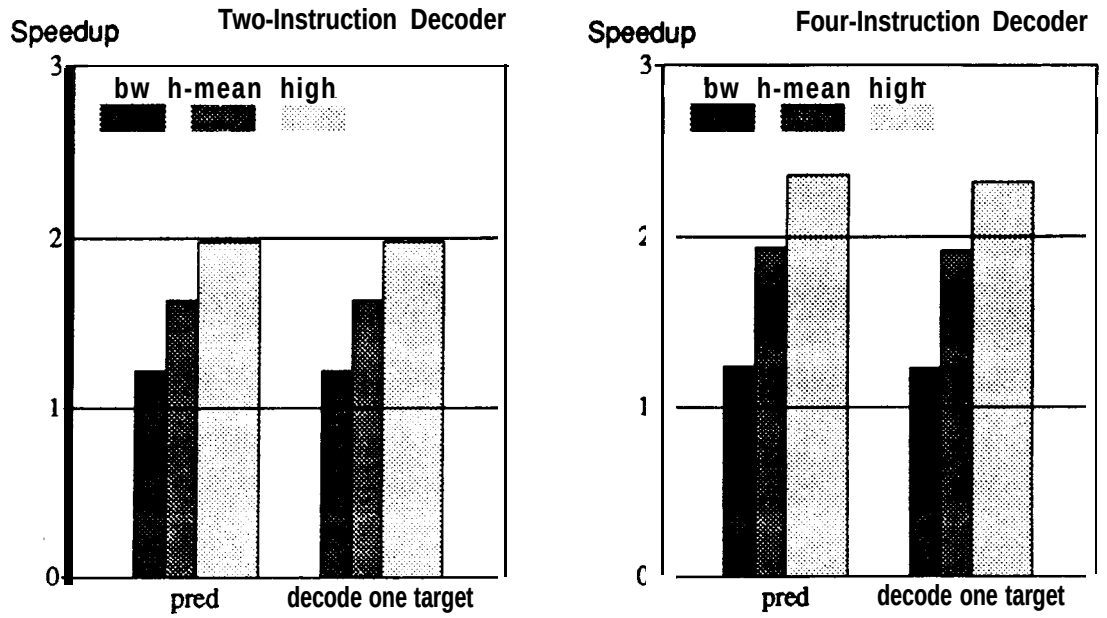

Figure 33. Performance Decrease Caused by Computing One Branch Target Address Per Decode Cycle

4.6 Observations and Conclusions

To sustain high instruction throughput, an instruction fetcher for a super-scalar processor must be able to fetch successive runs of instructions without intervening delay, and should have a wide (four-instruction) decoder. The instruction fetcher also should be able to fetch through as many as four runs of instructions before an unresolved branch can stall the decoder. Within this framework, software can help improve instruction-fetch efficiency by aligning and merging instructions runs, and by allocating decoder resources (specifically, register-file read ports) so that high performance is obtained with relatively simple hardware.

Hardware branch prediction can be added to the instruction cache for a small relative cost. Alternatively, software branch prediction incurs almost no hardware cost, and has the added advantage that software can, based on the software prediction, align and merge instruction runs. Although this study does not examine software branch prediction, aligning, and merging in detail, its results indicate that software branch prediction is about as accurate as hard: ware branch prediction, and that software aligning and merging can increase performance by about 5% (this improvement is in addition to other software scheduling to improve instruction independence). However, if software branch prediction is used, the instruction

fetcher must still be able to fetch instruction runs without intervening delays; the instruction-set architecture cannot have fully-general relative branches, and branches must be readily identifiable by fetch hardware.

A four-instruction decoder provides a higher sustained instruction bandwidth than a two-instruction decoder, and also allows a higher peak execution rate. With software support, the four-instruction decoder can be constructed without the eight register-file read ports that are implied by a simplistic implementation. A two-instruction decoder may be an adequate alternative if software techniques alone are used to schedule instructions. In the latter case, both the execution hardware and the decoder are simpler, though performance is lower. The decision between a two-instruction and a four-instruction decoder can also depend on the position of instruction cache with respect to the processor. If these are on separate chips, it may be too expensive to communicate four instructions in a single cycle. Throughout the remainder of this study, performance results are shown for both two-instruction and four-instruction decoders.

Since efficient instruction fetching relies on some sort of branch prediction, there must be a mechanism for undoing the effect of instructions executed along a mispredicted path. This mechanism, and its relationship to register renaming, is the topic of the following chapter.

Chapter 5
Operand Management

A super-scalar processor achieves high instruction throughput by fetching and issuing instructions under the assumption that branches are predicted correctly and that exceptions do not occur. This allows instruction execution to proceed without waiting on the completion of previous instructions. However, the processor must produce correct results even when these assumptions fail. Correct operation requires restart mechanisms for canceling the effects of instructions that were issued under false assumptions. Fortunately, the restart mechanism added to the processor can also support register renaming, improving the performance of out-of-order issue.

Sustaining a high instruction throughput also requires a high operand-transfer rate. High performance requires multiple result buses and bypass paths to forward results directly to waiting instructions. Moreover, the processor requires mechanisms to control the routing of operand values to instructions and to insure that instructions are not issued until all input operands are valid. This chapter examines the magnitude of this hardware and the performance it provides. Supplying operands for instructions is complicated by the fact that the processor, on every cycle, can be generating multiple, out-of-order results and attempting to prepare multiple instructions for issue.

Finally, this chapter explores the complexity of implementing restart, register renaming, and result forwarding. This hardware is complex, but there are no simple hardware alternatives that provide nearly the same performance.

5.1 Buffering State Information for Restart

The implementation of precise interrupts with out-of-order completion requires buffering so that the processor can maintain both the state required for computation and the state required for precise interrupts. This section describes the different types of state information to be maintained and describes four previously-proposed buffering techniques that maintain this information: checkpoint repair, a history buffer, a reorder buffer, and a future file. This section contrasts the four buffering techniques, and will show that either the reorder buffer or the future file is appropriate for restarting a super-scalar processor. Section 5.3 later considers the roles of the reorder buffer and future file in analyzing and enforcing data dependencies, and concludes that the reorder buffer is preferred.

5.1.1 Sequential, Look-Ahead, and Architectural State

To aid understanding of the mechanisms for precise interrupts, this section introduces the concepts of ***sequential, look-ahead,*** and ***architectural*** state, illustrated in Figure 34. Figure 34 shows the register assignments performed by a sequence of instructions: in this sequence, completed instruction are shown in boldface.

The sequential state is made up of the most recent assignments performed by the longest uninterrupted sequence of completed instructions. In Figure 34, the assignments performed by three of the first four instructions of the sequence are part of the sequential state (as are assignments performed by previous instructions not shown and assumed completed). The assignment to R7 in the second instruction does not appear in the sequential state because it has been superseded by the assignment to R7 in the fourth instruction. Though the sixth instruction is shown completed, its assignment is not part of the sequential state because it has completed out-of-order: the fifth instruction has not yet completed

All assignments starting with the first uncompleted instruction are part of the look-ahead state, and are shown italicized in Figure 34. The look-ahead state is made up of actual register values as well as pending updates, since there are both completed and uncompleted instructions (in the hardware, pending updates are represented by tags). Because of possible exceptions, all assignments are retained in the look-ahead state. For example, the assignment to R3 in the sixth instruction is not superseded by the assignment to R3 in the eighth instruction; both assignments should be considered part of the look-ahead state and added in proper order to the sequential state. To further illustrate how the look-ahead state is added to the sequential state, the assignments of the fifth and sixth instructions will become part of the sequential state as soon as the fifth instruction completes successfully, and at that time the

instruction sequence	items in sequential state	items in look-ahead state	items in architectural state
R3 := . . . (1)	**R3 := . . . (1)**		
R7 := . . . (2)			
R8 := . . . (3)	**R8 := . . . (3)**		
R7 := . . . (4)	**R7 := . . . (4)**		**R7 := . . . (4)**
R4 := . . . (5)		R4 := . . . (5)	R4 := . . . (5)
R3 := . . . (6)		*R3 := . . . (6)*	
R8 := . . . (7)		*R8 := . . . (7)*	*R8 := . . . (7)*
R3 := . . . (8)		*R3 := . . . (8)*	*R3 := . . . (8)*

Figure 34. Illustration of Sequential, Look-Ahead, and Architectural State

assignment to R3 in the sixth instruction will suppress the assignment to R3 in the first instruction.

The architectural state consists of the most recently completed and pending assignments for each register, relative to the end of the instruction sequence. This is the state **that** must be accessed by an instruction following this sequence, for correct operation. In the architectural state, the pending assignment to R3 in the eighth instruction supersedes the completed assignment to R3 in the sixth instruction, because a subsequent instruction must **get the** most recent value assigned to R3. If a subsequent instruction accessing R3 were **decoded** before the eighth instruction completed, the decoded instruction would obtain a tag for the new value of R3, rather than an old value for R3. Note that the architectural state is not separate from the sequential and look-ahead states, but is obtained by combining these states.

All hardware implementations of precise interrupts described in the following sections must correctly maintain the sequential, look-ahead, and architectural states. These implementations differ primarily in the mechanisms used to isolate and maintain these sets of state.

5.1.2 Checkpoint Repair

Hwu and Patt [1987] describe the use of checkpoint repair to recover from mispredicted branches and exceptions. The processor provides a set of ***logical spaces,*** where each logical space consists of a full set of software-visible registers and memory. Of all the logical spaces, only one is used for current execution. The other logical spaces contain backup copies of sequential state that correspond to some previous point in execution. A various times during execution, a checkpoint is made by copying the contents of the current logical space to a backup space. Logical spaces are managed as a queue, so making a checkpoint discards the oldest checkpointed state. The copied state is not necessarily the sequential state at that time, but the checkpointed state is updated, as instructions complete, to bring it to the desired sequential state. After the checkpoint is made, computation continues in the current logical space. Restart is accomplished, if required, by loading the contents of the appropriate backup logical space into the current logical space; the backup space used to restart state depends on the location of the fault with respect to the location of the checkpoint in the instruction sequence.

Hwu and Patt propose two checkpoint mechanisms: one for exceptions, and one for mispredicted branches. This approach is based on presumed differences between exception restart and misprediction restart, and is intended to reduce the number of logical spaces required. Exceptions can happen at any instruction in the sequence, but exception restart is

required infrequently. Thus, checkpoints for exception restart can be made infrequently and at widely-separated points in the instruction sequence. If an exception does occur, the processor recovers the state at the exception point by reloading the checkpoint preceding the exception point (this may be the state several **instructions before the point** of exception) and executing instructions sequentially up to the point of the exception. In contrast, **misprediction** occurs only at branch points, and misprediction restart is required frequently. Thus, checkpoints for **misprediction** restart are made at every branch point, and contain the precise state to restart execution immediately after the **mispredicted** branch.

To avoid the time spent copying state to the backup logical spaces, Hwu and Patt propose implementing the logical spaces with multiple-bit storage cells. For example, each bit in the register file might be implemented by four bits of storage: one bit for the current logical space and three bits for three backup logical spaces. This complexity is the most serious disadvantage of this proposal. There is a tremendous amount of storage for the logical spaces, but the contents of these spaces differ only by a few locations (depending on the number of results produced between checkpoints). It is much more efficient to simply maintain these state differences in a dedicated structure such as the reorder buffer described in Section 51.4. Because of this inefficiency, checkpoint repair is not an appropriate restart mechanism.

5.1.3 History Buffer

Figure 35 shows the organization of a ***history buffer***. The history buffer was proposed by Smith and Pleszkun **[1985]** *as* a means for implementing precise interrupts in a pipelined scalar processor with out-of-order completion. In this organization, the register file contains the architectural state, and the history buffer stores items of the sequential state which have been superseded by items of look-ahead state. The look-ahead state is not maintained separately, but is part of the state in both the register file and the history buffer.

The history buffer is managed as a FIFO. When an instruction is decoded, the current value of the instruction's destination register is copied to the history buffer. When a value reaches the head of the history buffer, it is discarded if the associated instruction (that is, the instruction that caused the value to be placed into the history buffer) completed successfully. If this instruction has not completed, the history buffer does not advance until the instruction does complete. If the instruction completes with an exception, all other pending instructions are completed, then all active values in the history buffer are copied-from tail to head-back into the register file. This restores the register file to the sequential state at the point of the exception. Values are copied from tail to head so that, if there are multiple values for the

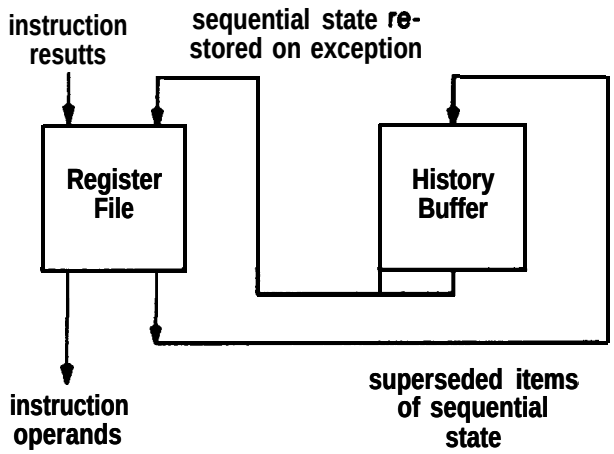

Figure 35. History Buffer Organization

same register, the oldest value will be placed into the register file last—other values are part of the look-ahead state.

Unfortunately, the history buffer has two significant disadvantages that are avoided by other schemes presented below. First, it requires additional ports on the register file for transferring the superseded values into the history buffer. For example, a four-instruction decoder would require as many as four additional ports for reading result-register values. These additional ports contribute nothing to performance. Second, the history buffer requires several cycles to restore the sequential state into the register file. These cycles are probably unimportant for exceptions, because exceptions are generally infrequent. However, the additional cycles are excessive for mispredicted branches. For these reasons, the history buffer is inappropriate for restart in a super-scalar processor.

5.1.4 Reorder Buffer

Figure 36 shows the organization of a ***reorder buffer*** [Smith and Pleszkun 1985]. In this organization, the register file contains the sequential state, and the reorder buffer contains the look-ahead state. The architectural state is obtained by combining the sequential and look-ahead states and ignoring all but the most recent updates to each register. Pleszkun et al. [1987] and Sohi and Vajapeyam [1987] demonstrate ways to unify the sequential and look-ahead states using software and associative hardware, respectively.

As with the history buffer, the reorder buffer is also managed as a FIFO. When an instruction is decoded, it is allocated an entry in the reorder buffer. The result value of the instruction is written into the allocated entry after the instruction completes. When the value reaches the

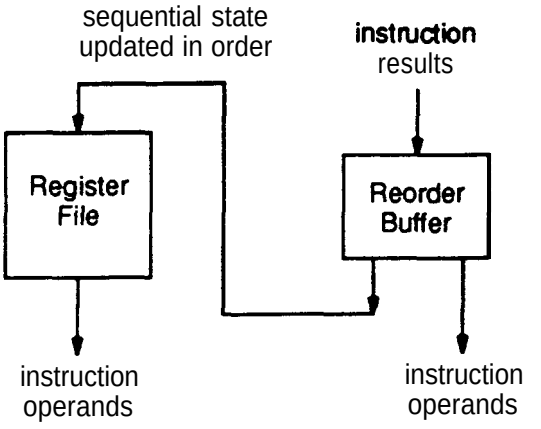

Figure 36. Reorder Buffer Organization

head of the reorder buffer, it is written into the register file, if there are no exceptions associated with the instruction. If the instruction is not complete when its entry reaches the head of the reorder buffer, the reorder buffer does not advance until the instruction is complete, but available entries may continue to be allocated. If there is an exception, the entire contents of the reorder buffer are discarded, and the processor reverts to accessing only the sequential state in the register file.

The reorder buffer has the disadvantage of requiring an associative lookup to combine the sequential and look-ahead states. Furthermore, this associative lookup is not straightforward, because it must obtain the most recent assignment if there is more than one assignment to a given register in the reorder buffer. This requires that the associative lookup be prioritized by instruction order, and that the reorder buffer be implemented as a true FIFO array, rather than as a circularly-addressed register array.

However, the reorder buffer overcomes both disadvantages of the history buffer. It does not require additional ports on the register file, and it allows some or all of the look-ahead state to be discarded in a single cycle. Although the reorder buffer appears to have more ports than are required to supply instruction operands, the additional ports are simply the outputs of the final entries of the FIFO, and do not have the costs associated with true ports (this consideration further argues for implementing the reorder buffer as a FIFO).

5.1.5 Future File

The associative lookup in the reorder buffer can be avoided by using a *future file* to contain the architectural state, as shown in Figure 37 (this future file is a slight variation of the future

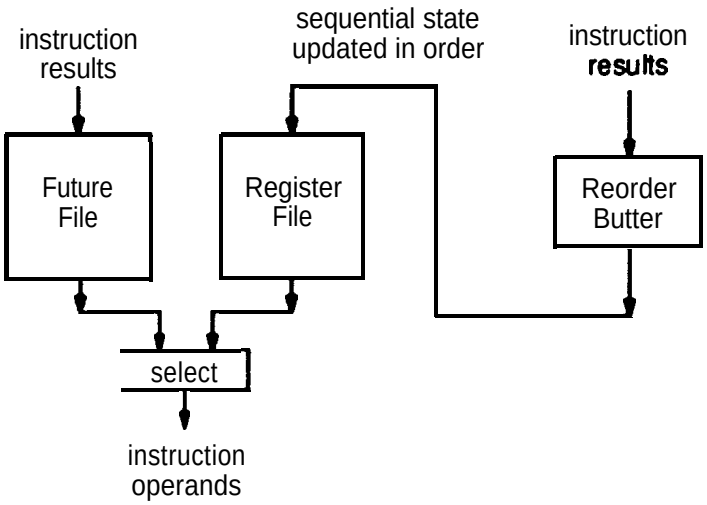

Figure 37. Future File Organization

file proposed by Smith and Pleszkun [1985]). In this organization, the register file contains the sequential state and the reorder buffer contains the look-ahead state. Both the reorder buffer and the register file operate as described in Section 5.1.4, except that the architectural state is duplicated in the future file. The future file can be structured exactly as the register file is, and can use identical access techniques.

With a future file, operands are not accessed from the reorder buffer. The reorder buffer only updates the sequential state in the register file as described in Section 5.1.4. During instruction decode, register identifiers are applied to both the register file and the future file. If the future file has the most recent entry for that register, that entry is used (this entry may be a tag rather than a value if the entry has a pending update); otherwise the value in the register file is used (values in the register file are always valid, and there ate no tags). Once a register in the future file has been marked as containing the most recent value (or a tag for this value), subsequent instructions accessing this register obtain the value in the future file (or a tag for this value). If an exception occurs, the contents of the future file are discarded (by being marked as not containing the most recent value), the contents of the reorder buffer are discarded or are written into the register file to complete all updates to the sequential state, and the processor reverts to accessing the sequential state in the register file.

When an instruction completes, its result value is written at the future-file location identified by the result register identifier (if this is the most recent update), and the value is also written into the reorder-buffer entry that was allocated during decode. At this point, the result value

in the future file has been marked as the most recent value, causing it to effectively replace the value in the register file. Once an entry is marked as most recent, it remains valid until the next exception occurs (it may still contain a tag instead of a value at various times during execution).

The future file described by Smith and Pleszkun [1985] is slightly different than the future file described here, because, in their organization, only the future file provides operands. Their scheme uses the register file to communicate the sequential state to an exception handler. The scheme described here allows both the future file and the register file to provide operands. This permits quick restart after a mispredicted branch, because the register file can supply all operands after restart, without any copying to the future file.

The future file overcomes the disadvantages of the history buffer without the associative comparators of the reorder buffer. However, this advantage is at the expense of an additional array (the future file) and validity and tag bits associated with entries of this array.

5.2 Restart Implementation and Effects on Performance

Given that the processor contains adequate buffering to restart execution at a previous version of the processor state, there are still the problems of identifying this version of state and either reinitiating execution or reporting the state to an exception handler. The processor should restart automatically after a mispredicted branch, to avoid excessive penalties for misprediction. To allow restart after an exception, the processor simply communicates a consistent sequential state to software.

5.2.1 Mispredicted Branches

Sohi and Vajapeyam [1987] mention, as a subject for future research, using a reorder buffer to restart after a mispredicted branch. In general, branch recovery is straightforward except for the proper handling of instructions preceding the mispredicted branch. Figure 38 shows the action required to correct the processor state if a mispredicted branch follows the sixth instruction in the sequence of Figure 34. The state must be backed up to the point of the mispredicted branch, but it is not correct to discard the entire look-ahead state, because some of this state is associated with incomplete instructions which preceded the branch. These items in the look-ahead state must be preserved. Furthermore, each previously-superseded value in the look-ahead state must be “uncovered” so that subsequent instructions obtain correct operands values.

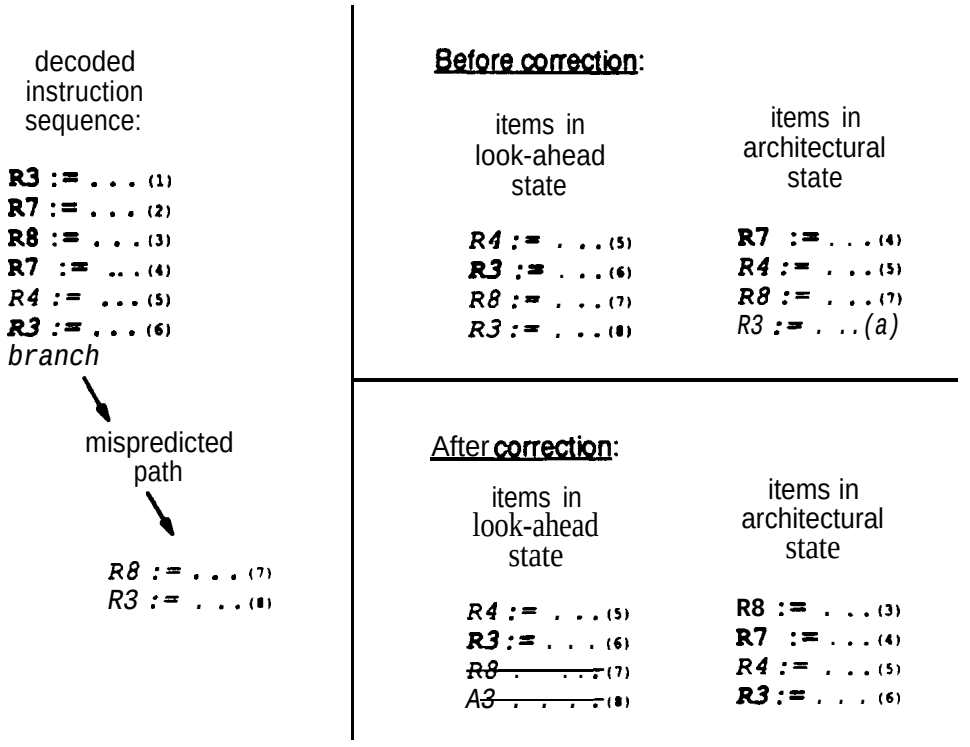

Figure 38. Correcting State After a Mispredicted Branch

If the architectural state is provided via a prioritized associative lookup in the reorder buffer, backing up the state after a mispredicted branch is a matter of clearing all entries that were allocated after the mispredicted branch. This requires the capability to reset a variable number of reorder-buffer entries, but is otherwise not complex to implement.

Backing up the state after a mispredicted branch is simpler if it is performed after the mispredicted branch point reaches the head of the reorder buffer. At this time, all instructions preceding the branch are committed to the sequential state, and restart involves only resetting the entire reorder buffer and/or future file, depending on the implementation. This technique is required with a future file because the future file stores only the architectural state. The portion of the look-ahead state required to partially back up the architectural state *on* a mispredicted branch is in the *reorder* buffer, but *is* inaccessible by instructions when a future file is used.

Figure 39 shows, for the sample benchmarks, the distribution of additional branch delay caused by waiting for a mispredicted branch to reach the head of the reorder buffer before restart. On the average, waiting adds no delay for about half of the mispredicted branches. The reason for this is that the outcome of a conditional branch depends on the results of other

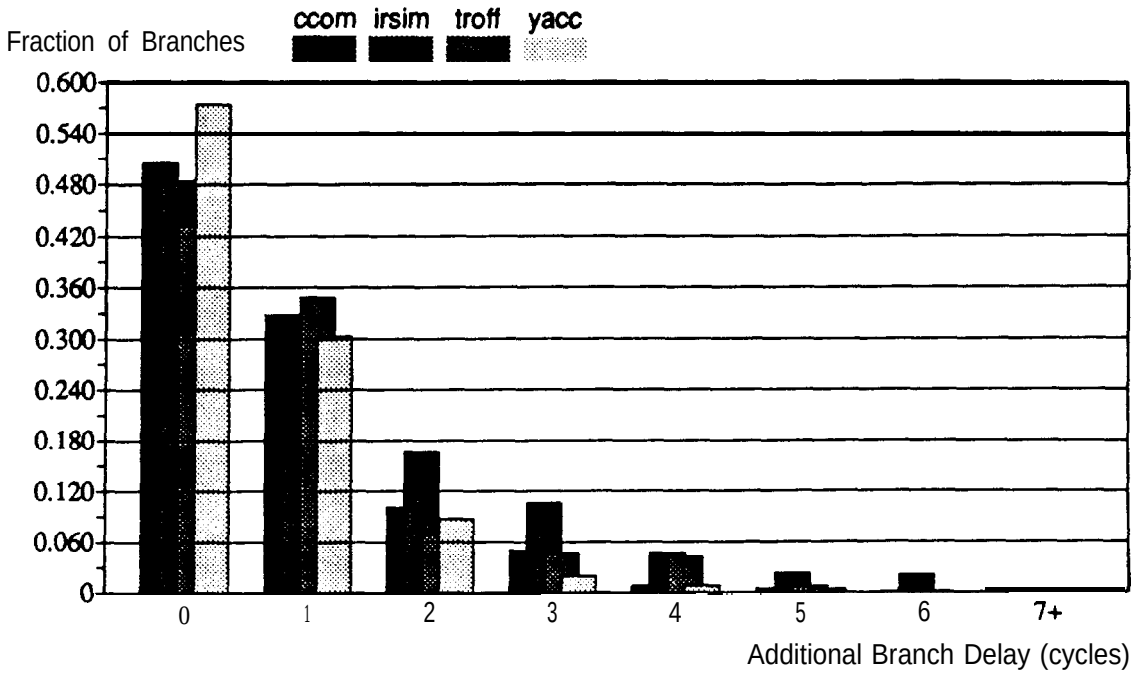

Figure 39. Distribution of Additional Branch Delay Caused by Waiting for a Mispredicted Branch to Reach the Head of the Reorder Buffer

instructions, and a branch is often at or near the end of the dependency chain. It is likely that the branch point will have reached the head of the reorder buffer by the time the outcome of the branch can be determined anyway, so there is only a small penalty for always waiting until the branch reaches the head of the reorder buffer. Figure 40 shows the impact of this additional penalty on performance (the penalty is the same regardless of whether or not a future file is used with the reorder buffer). Adding this small penalty to an already-large branch penalty (Figure 13, page 39) causes only a small proportional decrease in performance (4–5%).

Regardless of the mechanism used to restart after a mispredicted branch, there must be some mechanism for identifying the location of a branch point in the reorder buffer. The obvious way to identify this location is to allocate a reorder-buffer entry for each branch, even if the branch does not generate a result (a procedure call does write a return address into a register). When the processor detects a mispredicted branch, it invalidates all reorder-buffer entries subsequent to the entry allocated for the branch. The reorder-buffer entries can be invalidated selectively as soon as the misprediction is detected, or the entire reorder buffer (and, possibly, the future file) can be invalidated after the branch reaches the head of the reorder buffer.

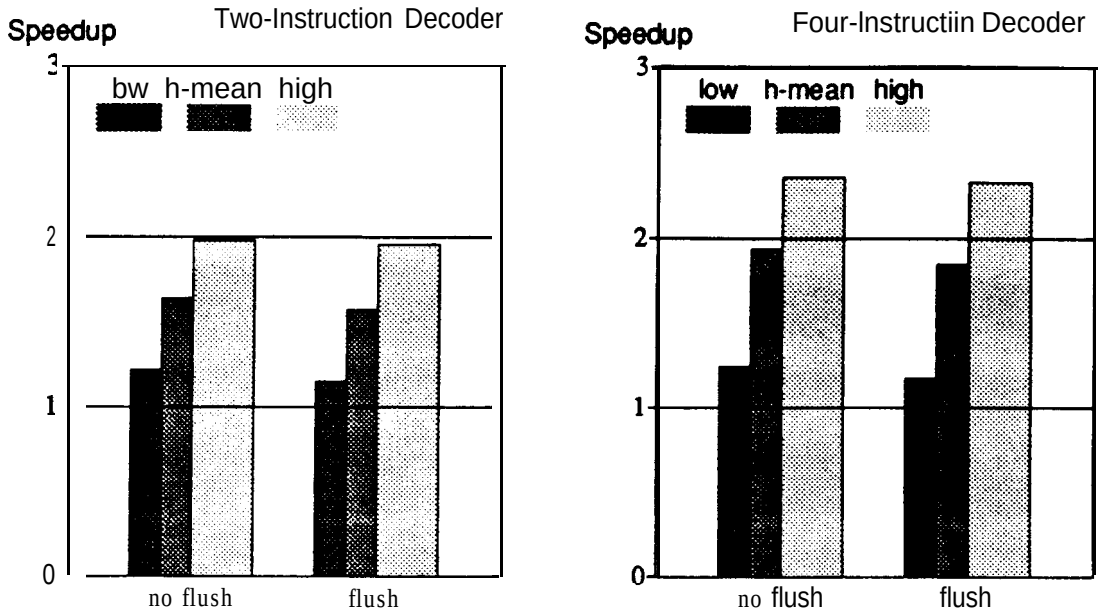

Figure 40. Performance Degradation Caused by Waiting for a Mispredicted Branch to Reach the Head of the Reorder Buffer

5.2.2 Exceptions

Providing a consistent sequential state for exception restart encounters a few complications beyond those encountered in restarting after mispredicted branches. First, when the restart occurs at any instruction, there must be a way to recover a program counter for restarting instruction execution (in the case of mispredicted branches, the correct program counter is obtained when the branch is executed). Second, there must be a way to determine the exception point in the reorder buffer (in the case of mispredicted branches, this point is marked simply by allocating reorder-buffer entries to branch instructions).

To allow a program counter to be easily recovered upon exception, Smith and Pleszkun [1985] proposed keeping program counter values of each instruction (determined during decode) in the reorder buffer along with the instruction result. This approximately doubles the storage in the result buffer. However, their proposal did not allocate reorder-buffer entries for branch instructions (because they were not concerned with restarting after mispredicted branches), and allocating entries for branches provides a convenient way to recover the program counter. Recovery is accomplished by setting a program-counter register as every branch point is removed from the reorder buffer, and incrementing the value (by one or two instructions) as non-branching instructions are removed from the reorder buffer. If an excepting instruction appears at the head of the reorder buffer, the program counter register

indicates the location of this instruction in memory. Note that maintaining a correct program counter requires that **all** instructions be allocated reorder-buffer entries, rather than just those that write registers. This is one reason for allocating a reorder-buffer entry for each instruction; other reasons are presented below.

Determining the location of excepting instructions in the reorder buffer requires that the reorder buffer be augmented with a few bits of instruction state to indicate whether or not the associated instruction completed successfully. This state information can also be used to indicate the location of branch points, for uniformity. Indicating exceptions in this manner also requires that each instruction be allocated a reorder-buffer entry, whether or not it writes a processor register. An instruction may create an exception even though it does not write a register (e.g. a store). Furthermore, because exceptions are independent of register results, 'the instruction-state information in the reorder buffer must be written independently of other results.

An additional advantage to allocating a reorder-buffer entry for every instruction is that it provides a convenient mechanism for releasing stores to the data cache so that the sequential state is preserved in memory [Smith and Pleszkun **1985].** By keeping identifiers for the store buffer in the reorder buffer, the reorder buffer can signal the release of store-buffer entries as the corresponding stores reach the head of the result buffer. If the store buffer is allocated during the decode stage, store-buffer identifiers are readily available to be placed into the reorder buffer. After being released in this manner, store exceptions (e.g. bus errors) are not restartable unless there is some other mechanism to allow restart.

It is also simpler for the decoder to allocate a reorder-buffer entry for every instruction than to allocate an entry just for those instructions which generate results. The decoder ***need*** only determine which instructions **are** valid (as determined during instruction fetch), rather than also determine which instructions generate results.

The essential advantage of allocating a reorder-buffer **entry** for every instruction is that it preserves instruction-sequence information even with out-of-order issue. This **information** is useful for a variety of purposes.

5.2.3 Effect of Restart Hardware on Performance

The limited size of the reorder buffer-whether or not it is used with a future file-can reduce performance. Instruction decoding stalls when there is no available reorder-buffer entry to receive the result of a decoded instruction. (The history buffer, if used, would have an

analogous effect in addition to the delay caused by restoring state from the history buffer.) To avoid decoder stalls, the reorder buffer should be large enough to accept results during the expected decode-to-completion &lay of most instructions.

Figure 41 shows the effects of reorder-buffer size on performance for a two-instruction decoder and a four-instruction decoder, respectively. The simulation model for the super-scalar processor has two reorder buffers—one for integer results (and for other instructions, such as branches, which do not have results) and one for floating-point results-in anticipation that the integer and floating-point units may be implemented on separate chips. Only the size of the integer reorder buffer was varied for these results; the floating-point reorder buffer was held constant at eight entries.

There is little reduction in performance with twelve or more reorder-buffer entries, but performance decreases markedly with smaller numbers of entries. Considering instructions that generate results, the size of the reorder buffer is determined by the delays in generating results: it is necessary to provide enough buffering that the decoder does not stall if a result is not yet computed when the corresponding reorder-buffer entry reaches the head of the reorder buffer. When the reorder buffer is allocated for instructions without results (primarily stores and branches), the incremental demand placed on the reorder buffer by these instruc-

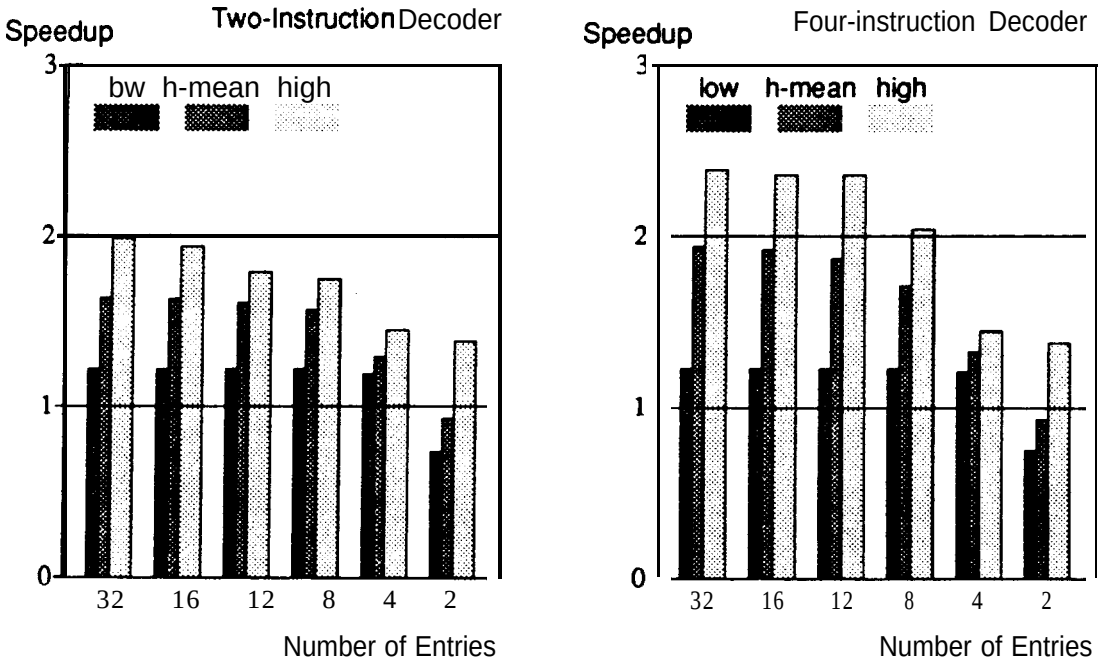

Figure 41. Effect of Reorder-Buffer Size on Performance: Allocating for Every Instruction

tions is directly proportional to their frequency relative to the result-generating instructions. The reorder-buffer size determined by the peak demand of result-generating instruction is still adequate if the reorder buffer is allocated for all instructions. *When the* peak demand of result-generating instructions is achieved, there is, by implication, a local reduction in the number of instructions without results, and vice versa.

Figure 41 also shows that, if the reorder buffer is too small, performance of the super-scalar processor can be less than the performance of the scalar processor. The reorder buffer causes the decoder stage to be dependent on the write-back stage, and prevents the processor pipeline from operating smoothly when the reorder buffer becomes full. The scalar processor does not suffer this effect because the processor model used for these results does not implement precise interrupts, even though it completes instructions out-of-order.

5.3 Dependency Mechanisms

If the compiler cannot insure the absence of storage conflicts, the hardware must either implement register renaming to remove storage conflicts or use another mechanism to insure that anti- and output dependencies are enforced. For this reason, it is useful to evaluate implementations of renaming and to compare renaming to other dependency-resolution mechanisms in terms of hardware, complexity, and performance.

This section describes the advantages of renaming, and shows how renaming is implemented with a reorder buffer and a future file. It also explores other proposed dependency mechanisms. Measurements indicate the relative importance of various hardware features in an attempt to identify areas where the logic can be simplified or where performance can be improved.

Most of the literature focuses on complete algorithms that avoid or enforce dependency constraints. This section evaluates prior proposals in terms of the primitive mechanisms for handling anti- and output dependencies. This approach suggests another alternative, here called ***partial renaming,*** that has nearly all of the performance of renaming. Despite this new alternative, this section argues that—if dependency hardware is required for best performance—renaming with a reorder buffer is the most desirable alternative.

5.3.1 Value of Register Renaming

Register renaming eliminates storage conflicts between instructions and thus increases instruction independence. Figure 42 shows the performance of the super-scalar processor

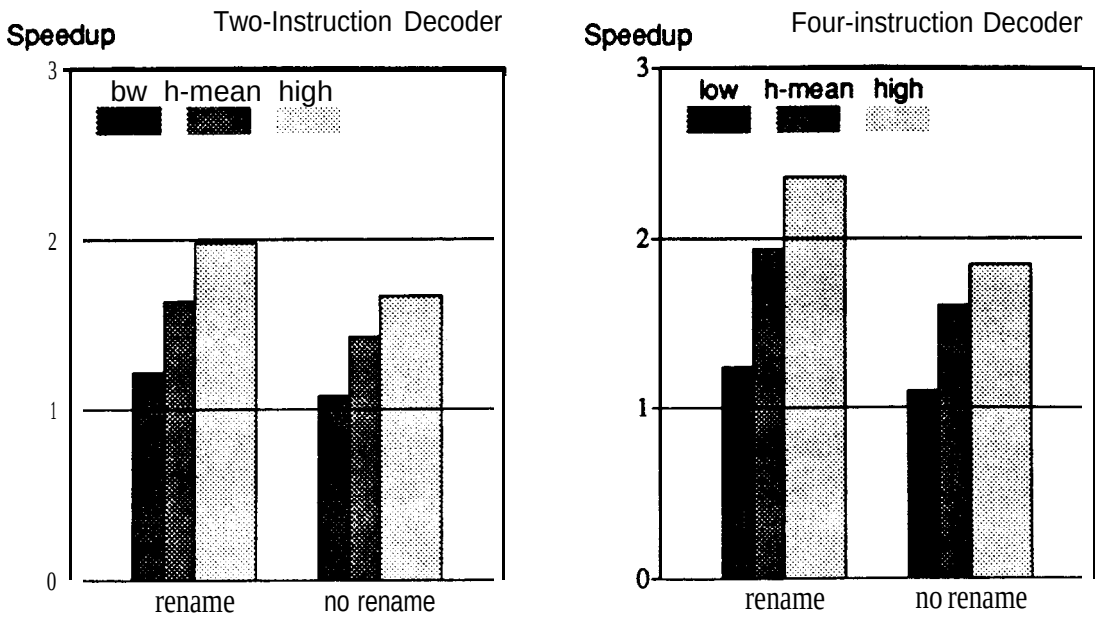

Figure 42. Reducing Concurrency by Eliminating Register Renaming

with and without register renaming. For these results, the processor uses Weiss and Smith's [1984] variation of Thor-ton's algorithm [Thotton 1970] to resolve dependencies in the "no rename" case: instruction decoding is stalled whenever a decoded instruction will create more than one instance of a register value. Eliminating renaming reduces performance by about 15% with a two-instruction decoder and by about 21% with a four-instruction decoder. It should be emphasized that these results are based on code generated by a compiler that performs aggressive register allocation, and this exaggerates the advantage of register renaming because registers are often reused. An architecture with a larger number of registers and different compiler technology would not experience the same advantage, but the degree to which this is true is unknown.

5.3.2 Register Renaming with a Reorder Buffer

A reorder buffer that uses associative lookup to implement the architectural state (Section 5.1.4) provides a straightforward implementation of register renaming. The associative lookup maps the register identifier to the reorder-buffer entry as soon as the entry is allocated, and the lookup is prioritized so that only the value for the most recent assignment is obtained (a tag for this value is obtained if the result is not yet available). There can be as many instances of a given register as there are reorder-buffer entries, so there are no storage conflicts between instructions. The values for the different instances are written to the

register file in sequential order. When the value for the final instance is written to the register file, the reorder buffer no longer maps the register, and the register file contains the only instance of the register.

5.3.3 Renaming with a Future File: Tomasulo's Algorithm

Tomasulo's algorithm [Tomasulo 1967] implements renaming by associating a tag with each register that has at least one pending update. This tag identifies the most recent value to be assigned to the register (in Tomasulo's implementation, the tag was the identifier of the reservation station containing the assigning instruction, but, as Weiss and Smith [1984] point out, the tag can be any unique identifier). When an instruction is decoded, it accesses register tags along with the contents of the operand registers. If the register has one or more pending updates, the tag identifies the update value required by the decoded instruction. Once an instruction is decoded, other instructions may overwrite this instruction's source operands without being constrained by anti-dependencies. Output dependencies are handled by preventing result writing if the associated instruction does not have a tag for the most recent value. Both anti- and output dependencies are handled without stalling instruction issue.

Tomsulo's algorithm is easily adapted to an implementation using a future file by placing the tag logic into the future file rather than in the register file. The future file requires that the tag array have four write ports which are separate from the result write ports, because tags are written immediately after decode rather than after instruction completion. Also, the tag array has two read ports so that tags may be compared before write-back, to prevent the writing of an old value. Finally, this implementation requires storage for the future file as well as storage for the reorder buffer.

5.3.4 Other Mechanisms to Resolve Anti-Dependencies

This section explores alternative mechanisms for handling anti-dependencies, in an attempt to identify a dependency mechanism that is simpler than renaming. There are two approaches to resolving anti-dependencies. First, anti-dependencies can be enforced by delaying the issue of any instruction that might overwrite the source operands of unissued instructions. Tomg [1984] and Acosta et al. [1986] describe such an approach using an instruction window called a ***dispatch stack.*** Alternatively, anti-dependencies can be avoided altogether by copying operands to dedicated buffers (e.g. reservation stations) during decode so that they cannot be overwritten by other register updates [Weiss and Smith 1984].

Of these two alternatives, copying operands is preferred, because it allows register accesses to be performed only once, in parallel with dependency checking. Register access is not repeated later when the instruction is issued, as it is in the first approach. The primary motivation of holding issue in the first approach is to avoid the additional storage to hold copies of operands; however, accessing registers just before instruction issue introduces additional issue latency and reduces performance. In addition, delaying issue for anti-dependencies requires comparing destination-register identifiers to all source-register identifiers of previous instructions. This comparison is somewhat costly and unnatural (Dwyer and Tomg [1987] give a good illustration), considering the number of combinations of source and destination operands to be compared and the need to release dependencies using source-register identifiers as a trigger rather than destination-register identifiers. These considerations outweigh any concern over the extra storage for operand copies.

It should be noted that copying operands simplifies the implementation of renaming [Tomasulo 1967] but is not sufficient to implement renaming because it does not enforce output dependencies. Copying operands simplifies renaming because it permits register access to be performed only once, and it permits a renaming mapping to be discarded immediately when the associated register must be renamed again. Without operand copying, it would be necessary to track each use of the old mapping and discard the mapping only after there were no remaining uses. Note that, in some cases, result forwarding (Section 5.5) supplies the operand for the old mapping after the mapping is discarded.

Any approach involving the copying of operands must correctly handle the situation where the operand value is not available when it is accessed. Stalling the decoder in this situation greatly reduces performance (this approaches the performance with in-order issue), so a more common solution is to supply a tag for the operand rather than the operand itself. This tag is also associated with the corresponding result value, so that the operand is obtained by forwarding when the result is eventually written. If there can be only one pending update to a register, the register identifier can serve as a tag. If there can be more than one pending update to a register (allowing more than one pending update avoids stalling the decoder for output dependencies and yields higher performance), there must be a mechanism for allocating result tags and insuring uniqueness.

5.3.5 Other Mechanisms to Resolve Output Dependencies

This section explores alternative mechanisms for handling output dependencies, again in an attempt to identify a dependency mechanism that is simpler than renaming. As with anti-dependencies, there are two approaches to resolving output-dependencies. First, output-

dependencies can be enforced by stalling decode when a decoded instruction might overwrite the destination register of an uncompleted instruction [Thorton 1970]. An alternative to stalling the decoder is to annotate the instruction with output-dependency information and place it into the reservation station or instruction window, allowing subsequent instructions to be decoded. The dispatch stack of Tomg [1984] and Acosta et al. [1986] uses the latter approach.

Stalling the decoder to resolve output dependencies insures that there is only one pending update to a register, but stalling the decoder in this situation reduces performance by 15–20%. The dispatch stack does not stall the decoder, but rather enforces output dependencies because it cannot issue any instructions that write a destination register out-of-order with respect to another write of the same register. Figure 43 compares the performance of this approach, with the set of bars labeled "no stall," in relation to the performance shown in Figure 42. Eliminating the decoder stalls for output dependencies improves performance, but renaming still has an advantage of about 6% with a two-instruction decoder and about 15% with a four-instruction decoder.

Unfortunately, the dispatch stack does not reduce complexity relative to renaming in return for the reduction in performance relative to renaming. Both the dispatch stack and renaming require mechanisms to allocate and track multiple updates to registers. Renaming has the

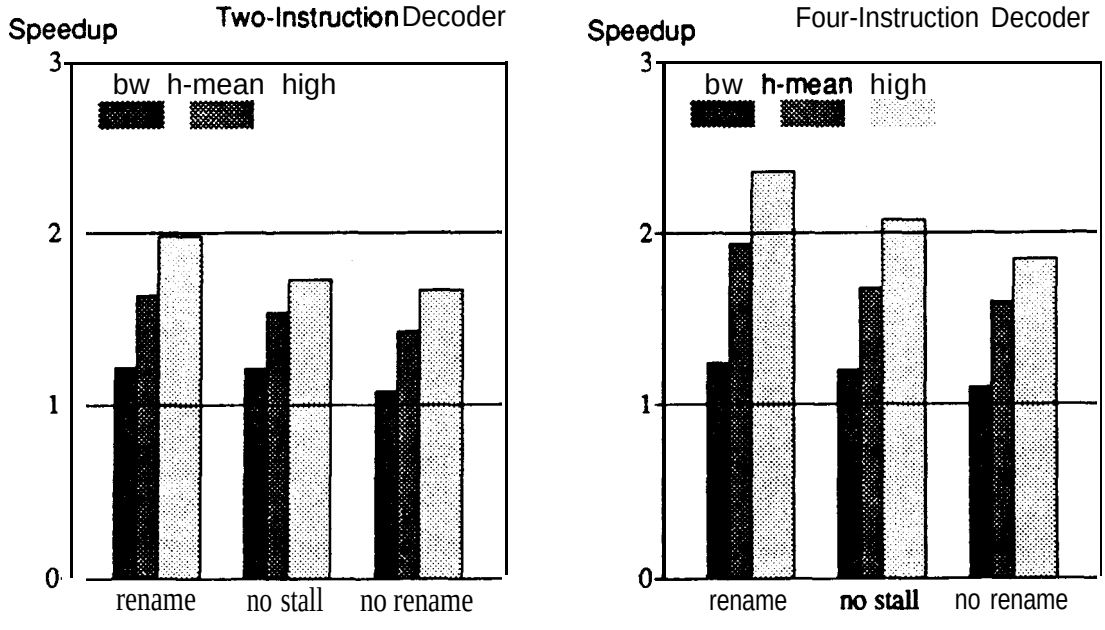

Figure 43. Performance Advantage of Eliminating Deco&r Stalls for Output Dependencies

advantage, though, that it does not require mechanisms to detect output dependencies and hold instructions until these dependencies are resolved. The dispatch stack requires even more associative logic than renaming to check dependencies (because of the number of dependencies checked). Renaming requires additional storage for renamed registers, but the reorder buffer supplies this storage handily, as well as allowing restart after exceptions and mispredicted branches.

5.3.6 Partial Renaming

The dependency mechanisms described so far suggest a technique that has not been proposed in the published literature. This technique is suggested by noting that the greatest performance disadvantage of Thorton's algorithm is due to stalling the decoder for output dependencies, and that the greatest performance disadvantage of the dispatch stack is due to holding instruction issue for anti-dependencies. An alternative to both approaches is allow multiple pending updates of registers to avoid stalling the decoder for output dependencies, but to handle anti-dependencies by copying operands (or tags) during decode. An instruction is not issued until it is free of output dependencies, so each register is updated in sequential order (although instructions still compete out-of-order). This alternative has almost all of the capabilities of register renaming, lacking only the capability to issue instructions so that register updates occur out-of-order. This alternative is called ***partial renaming*** because it is so close in capability to register renaming.

Figure 44 shows the performance of this partial renaming (with the set of bars labeled "partial"), in relation to the alternatives shown in Figure 43. Of all alternatives shown, this comes closest in performance to register renaming. Compared to renaming, there is a 1% performance reduction for a two-instruction decoder and a 2% performance reduction for a four-instruction decoder. The reason the disadvantage is slight is that out-of-order update is not very important, because there is some sequential ordering on the instructions. It is likely that register updates will occur in order anyway, and it is more likely that there are instructions waiting on a older register value than a newer register value. Forcing registers to be updated in or&r by stalling issue incurs little performance penalty.

Still, partial renaming must track multiple pending updates to registers, and, to correctly handle true dependencies, requires logic to allocate and distribute tags. The tag logic consists of an array with the number of entries equal to the number of registers. The tag entries are written during the cycle after decode with all result tags that were allocated during decode (priority logic is required if there can be more than one update of the same register decoded in a single cycle). This requires a number of write ports equivalent to the number of

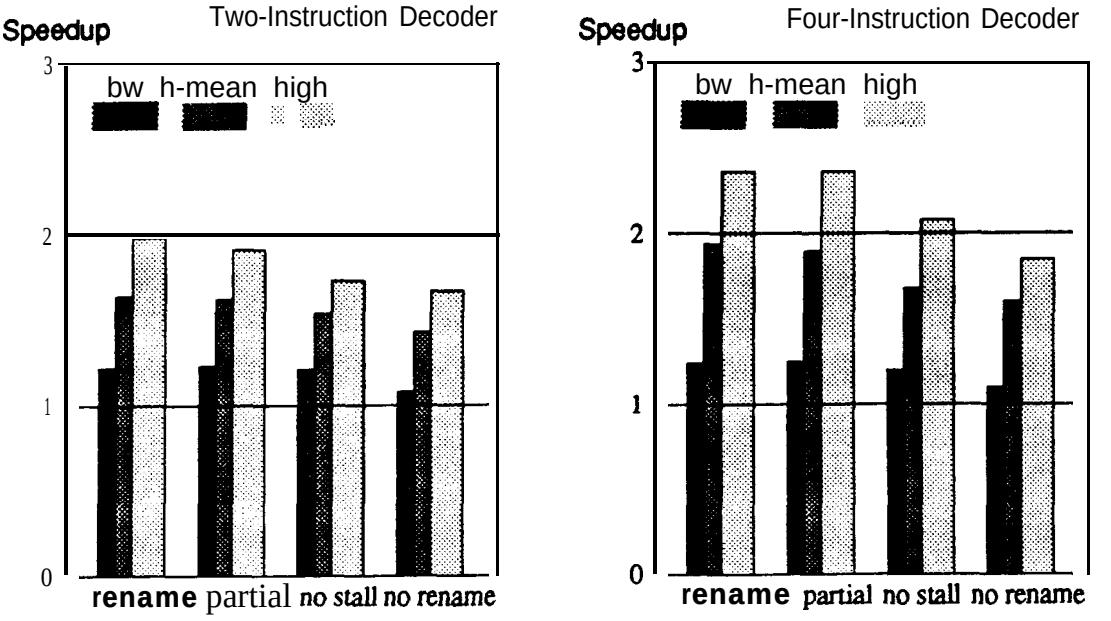

Figure 44. Performance Advantage of Partial Renaming

decoded instructions. During decode, the tag array is accessed using operand-register identifiers to obtain tags for operand values, ***and*** the tag array is accessed using result-register identifiers to obtain tags for the most recent result values. The latter tags are used to delay instruction issue if there is a pending update to the register, resolving output dependencies. Accessing tags thus requires a number of read ports equal to the number of operands read during **decode** ***plus*** the number of results of decoded instructions. Also, delaying instruction issue for output dependencies requires an additional comparator in each reservation-station entry to detect when a pending update is complete and that the instruction in the **reservation**-station entry is free of output dependencies.

With a four-instruction decoder, the processor organization described in Section 3.3.2 requires a 32-entry tag array (each entry consists of at least a four-bit tag and one valid bit), with four write ports and eight read ports, and 34, four-bit comparators in the reservation stations to implement partial renaming. This hardware is cumbersome, and its object is only to avoid associative lookup in the reorder buffer. Since partial renaming has lower performance and requires more hardware than renaming, it has little advantage over renaming with a reorder buffer.

5.4 Result Buses and Arbitration

For all results presented so far, two result buses have been used to carry results from the integer and floating-point functional units to the respective reorder buffers. Even at high levels of performance, the utilization of two result buses is about 50–55%, as illustrated in Figure 45. Figure 45 shows, for various numbers of integer result buses, the average bus utilization and average number of waiting results for the sample benchmarks. These results were measured for a processor with an average speedup of two. The principle effect of increasing the number of buses beyond two is to reduce short-term bus contention, as indicated by the reduction in the average number of results waiting for a result bus. However, as Figure 46 shows, eliminating this contention completely has a negligible performance benefit (less that 1% going from two to three buses).

In the processor model used in this study, the functional units request use of a result bus one cycle in advance of use. An arbiter decides, in any given cycle, which functional units will be allowed to gate results onto the result buses in the next cycle. There are two separate arbiters—one for integer results and one for floating-point results. Priority is given for result-bus use to those requests that have been active for more than one cycle, then to requests that have become active in the current cycle. The integer functional units are prioritized, in decreasing order of priority, as follows: ALU, shifter, branch (return addresses), loads. The

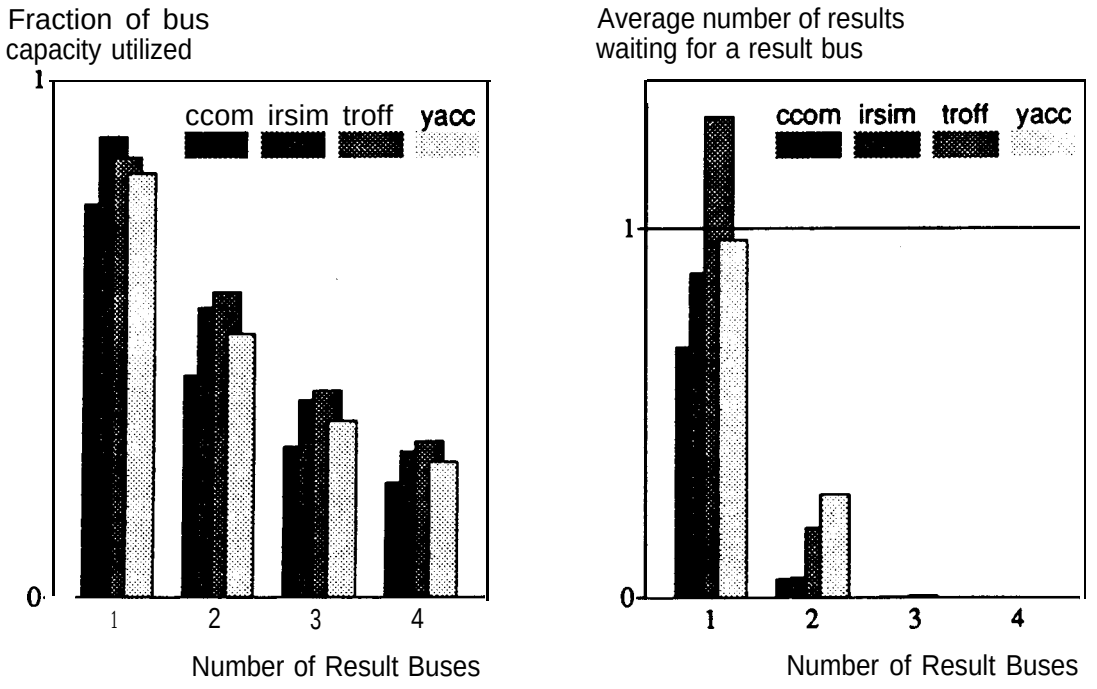

Figure 45. Integer Result-Bus Utilization at High Performance Levels

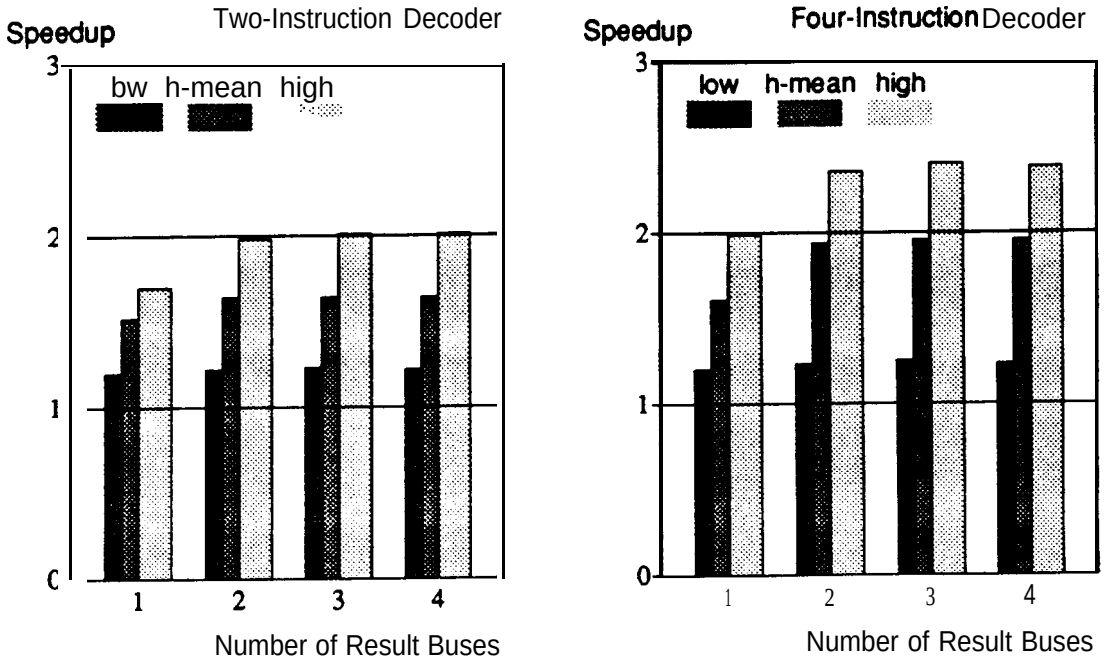

Figure 46. Effect of the Number of Result Buses on Performance

floating-point functional units are prioritized as follows: add, multiply, divide, convert. Prioritizing old requests over new ones helps prevent starvation, with a slight increase in arbiter complexity. The arbiter not only decides which functional units are to be granted use of the result buses, but also which bus is to be used by which functional unit.

If a functional unit requests a result bus but is not granted use, instruction issue is suspended at that functional unit until the bus is granted. Thus, a functional unit suffers start-up latency after experiencing contention for a result bus; this additional latency prevents result buses from being 100% utilized even when mote than one result, on the average, is waiting for a bus (Figure 45). The advantage of this approach, though, is that functional-unit pipeline stages can be clocked with a common signal, without the complication to the clocking circuitry caused by allowing earlier pipeline stages to operate while later stages are halted. Figure 46 indicates that reducing the effect of bus contention-by adding a third **bus**— yields a negligible improvement in performance. Hence, reducing the effects of bus contention by more exotic arbitration or additional pipeline buffering is unwarranted

5.5 Result Forwarding

Result forwarding supplies operands that were not available during decode directly to waiting instructions in the reservation stations. This resolves true dependencies that could not be

resolved during decode. Forwarding avoids the latency caused by writing, then reading the operand from a storage array, and avoids the additional ports that would be required to allow the reading of operands both during decode and before issue. Also, as previously discussed in Section 5.3.4, result forwarding simplifies the deallocation of result tags by satisfying all references to a tag as soon as the corresponding result is available. Figure 47 shows, for the sample benchmarks, the distribution of the number of operands supplied by forwarding per completed result. The horizontal axis is a count of the number of reservation-station entries receiving the result as an input operand. About two-thirds of all results are forwarded to one waiting operand, and about one-sixth of all results are forwarded to more than one operand. The high proportion of forwarded results indicates that forwarding is an important means of supplying operands.

The primary cost of forwarding is the associative logic in each reservation-station entry to detect, by comparing result tags to operand tags, that a required operand is on a result bus. The processor organization in Section 3.3.2 requires 60, four-bit comparators in the reservation stations to implement forwarding (load instructions have only one register-based operand, and the size of the reorder buffer fixes the size of the result tags).

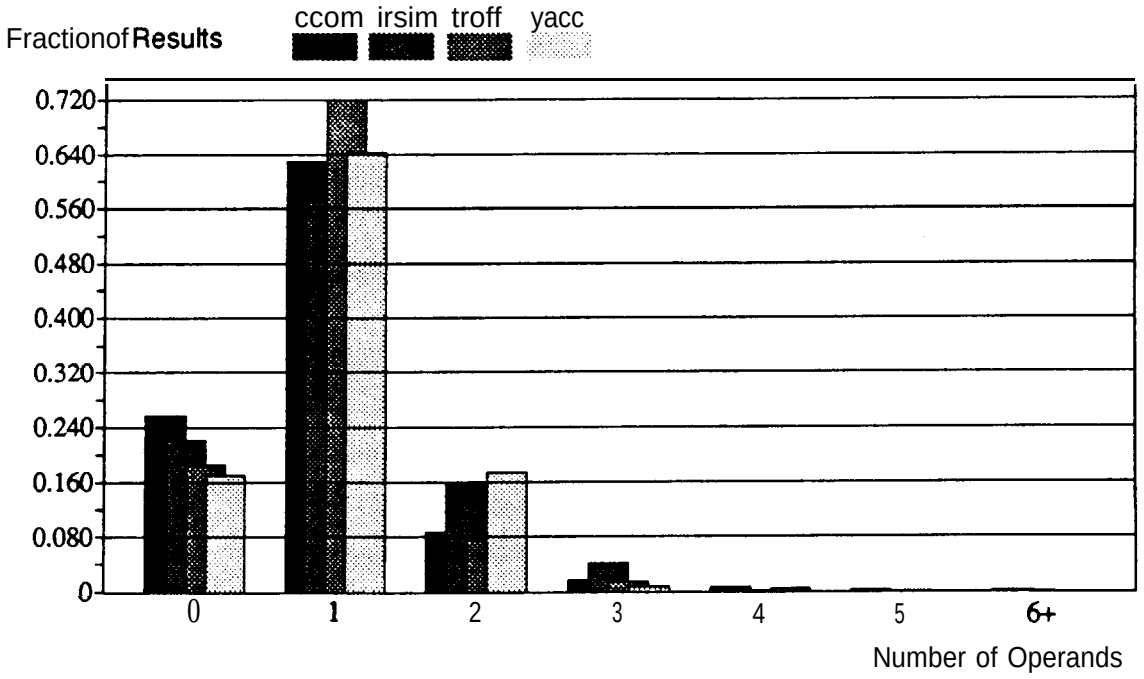

Figure 47. Distribution of Number of Operands Supplied by Forwarding, Per Result

The ***direct tag search*** proposed by Weiss and Smith [**1984]** eliminates the associative logic in the reservation stations. In the most general implementation, direct tag search maintains-for each result tag-a list of operands which are waiting on the corresponding result (Weiss and Smith proposed a single operand rather than a list, but a list is more general, because it allows a result to be forwarded to more than one location). Each operand is identified by the reservation-station location allocated to receive the operand When an instruction depends on an unavailable result, the appropriate reservation-station identifier is placed on the list of waiting operands. If the list is **full,** the decoder **stalls.** When the result becomes available, its tag is used to access the list, and the result is forwarded directly to the waiting instructions in the reservation stations.

Figure 48 indicates the performance of direct tag search for various numbers of list entries. For best performance, at least three entries are required for each result. Unfortunately, any implementation with more than one entry encounters difficulty allocating entries **and detect-**ing that the list is full. Furthermore, even an implementation with one list entry requires a storage array for the list table. In the processor described in Section 3.3.2, this table would have sixteen entries of six bits each, with four ports for writing reservation-station identifiers and two ports for reading reservation-station identifiers. There would also be sixty decoding

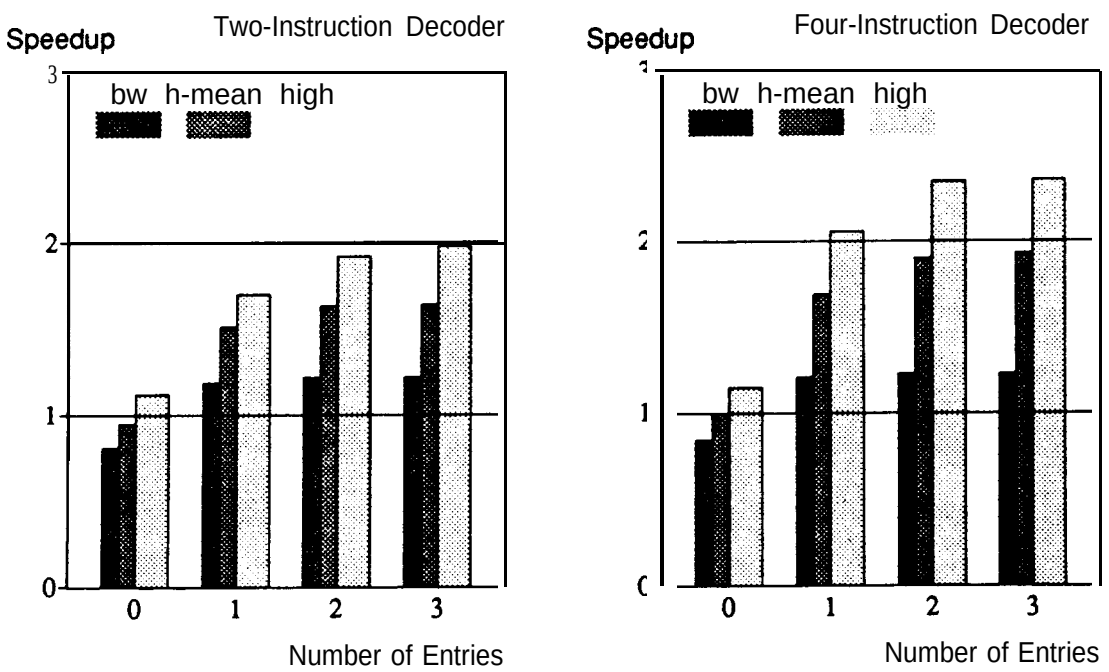

Figure 48. Performance of Direct Tag Search for Various Numbers of List Entries

circuits to gate the result into the proper reservation-station entry. Direct tag search was proposed for a scalar processor and does not save hardware in the super-scalar processor, because of the higher number of instructions decoded and completed per cycle.

In Figure 48, the performance with zero list entries is the performance that would result if forwarding were not implemented. These results show that the concurrency possible in the super-scalar processor does not compensate for the additional latency caused by a lack of forwarding, further illustrating the value of **forwarding.**

Forwarding not only takes a large amount of hardware, but also creates long critical &lay paths. Once a result is valid on a result bus, it is necessary to detect that this is a required result, and gate the result to the input of the functional unit so that the functional unit can begin operation as the result is written into the reorder buffer. To avoid having the forwarding delay appear as part of the functional-unit delay, and to allow data paths to be set up in time, the actual tag comparison should be performed at the end of the cycle preceding result availability. This in turn implies that the tags for the two results should appear on the tag buses a cycle before the results appear on the result buses. Providing result tags early is difficult, because it requires that result-bus arbitration be performed very early in the cycle.

5.6 Implementing Renaming with a Reorder Buffer

Previous sections have argued that a reorder buffer should be used to implement restarting and renaming, and have argued that forwarding is required to sustain performance. This section considers, in detail, a straightforward implementation of the instruction decoder, reorder buffer, reservation stations, and forwarding logic in light of the requirements of register renaming, operand routing, and restart. A four-instruction decoder is described, but the changes necessary for a two-instruction decoder should be obvious. The implementation presented here provides a good illustration of the hardware complexity, showing that the best alternative is quite complex.

5.6.1 Allocating Processor Resources

During instruction decode, the processor allocates result tags, reorder-buffer entries, and reservation-station entries to the instructions being decoded. Allocation is accomplished by providing instructions with identifiers for these resources, using the hardware organization shown in Figure 49. Hardware similar to that shown in Figure 49 is used for each resource to be allocated; for example, a version of this hardware is required for each reservation station, to allocate entries to instructions.

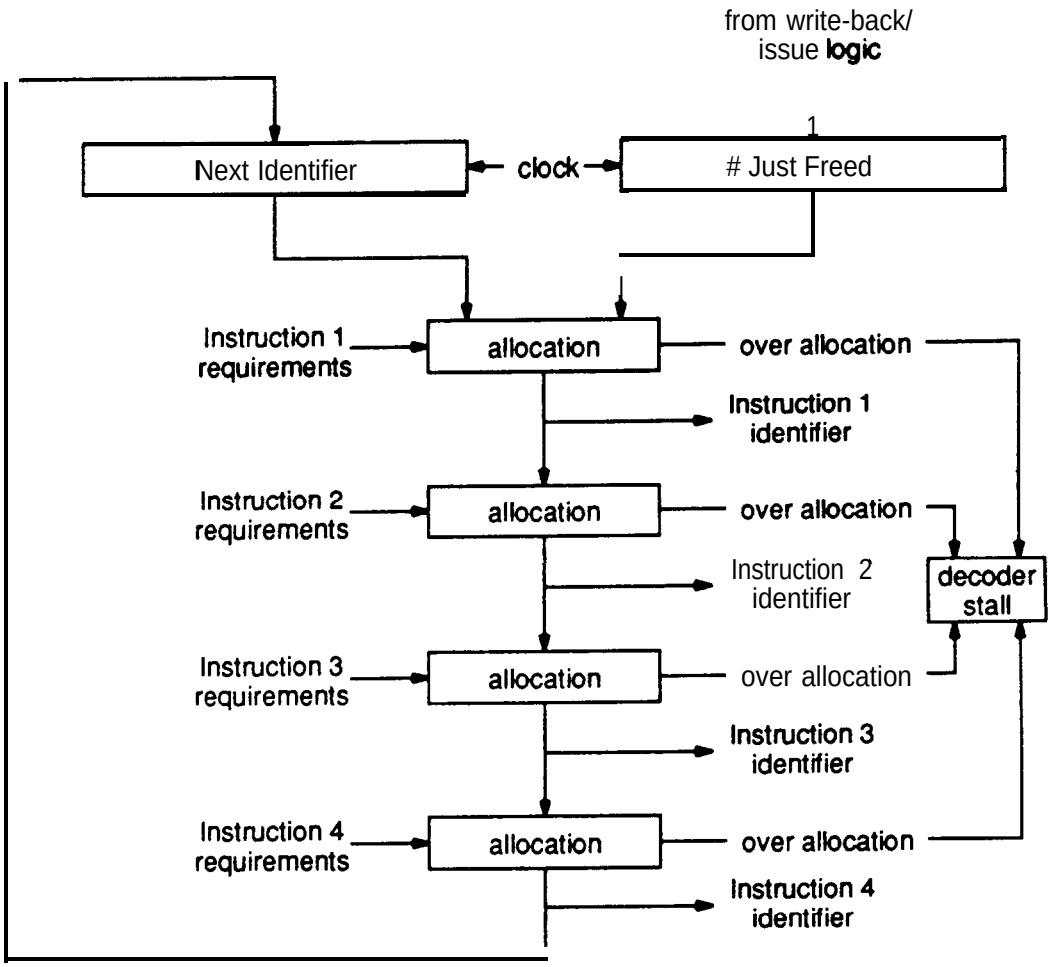

Figure 49. Allocation of Result Tags, Reorder-Buffer Entries, and Reservation-Station Entries

The allocation hardware has multiple stages, with one stage per instruction (these stages are not pipelined-they all operate in a single cycle). This organization takes advantage of the fact that resource identifiers are small (for example, a sixteen-entry reorder buffer requires four-bit entry identifiers), and of the the fact that allocation has nearly a full processor cycle to complete. The input to the first stage is an identifier for the first available resource. If the first instruction requires the resource, the first allocation stage uses this identifier and also forms an identifier for the next subsequent resource that is passed to the second stage (this next resource is assumed to be available). If the first instruction does not require the resource, the identifier is passed to the second stage unmodified. For example, in the case of reorder-buffer allocation, the first instruction would receive the identifier for the next avail-

able reorder-buffer entry, and the second instruction would receive the identifier for the second available reorder-buffer entry.

The identifier for the **first** available resource is based upon the resources allocated and freed in the previous cycle. For example, again in the case of reorder-buffer allocation, assume that the fifth reorder-buffer entry were allocated to the final instruction in the previous decode stage, and that two reorder-buffer entries were written into the register file (thus freed). The first valid instruction in the next decode cycle would be allocated the fourth entry (the previously-allocated entries are moved up two locations at the end of the same cycle), and other instructions would be allocated the fifth and subsequent entries.

Each allocation stage--other than the first **one**—is essentially an incrementer for resource identifiers. The stage either increments the identifier or passes it unmodified, depending on the needs of the corresponding instruction. If any incrementer attempts to allocate beyond the last available resource, the decoder is stalled, and the associated instruction (and all subsequent instructions) repeat the decode process on the next cycle. Note that the incrementers need not generate identifiers in numerical sequence, because the identifiers are used by hardware alone. The identifiers can be defined so that the incrementers take minimal logic.

In this implementation, there is separate hardware to allocate result tags for instructions. Reorder-buffer identifiers are inappropriate to use as result tags, because these identifiers are not unique. Reorder-buffer entries are allocated at the end of a FIFO, and it is likely that the same entries will be allocated over several decode cycles. However, the result tags have the same number of bits as reorder-buffer identifiers, because both track pending updates.

5.6.2 Instruction Decode

During instruction decode, the dependency logic must insure that all instructions receive operands or tags. In addition, the state of the dependency logic must be updated so that subsequent instructions obtain operands correctly. Figure 50 illustrates some of the dependency operations performed during instruction decode, focusing on the operations performed on the reorder buffer.

To obtain operands, the reorder buffer is searched associatively using the source-register identifiers of the decoded instructions. These source-register identifiers are compared to result-register identifiers stored in the reorder buffer, at each of four read ports. If the result is not available, a result tag is obtained. As Figure 50 shows, result tags use the same storage as result values, so that operand values and tags are treated uniformly and use the same buses

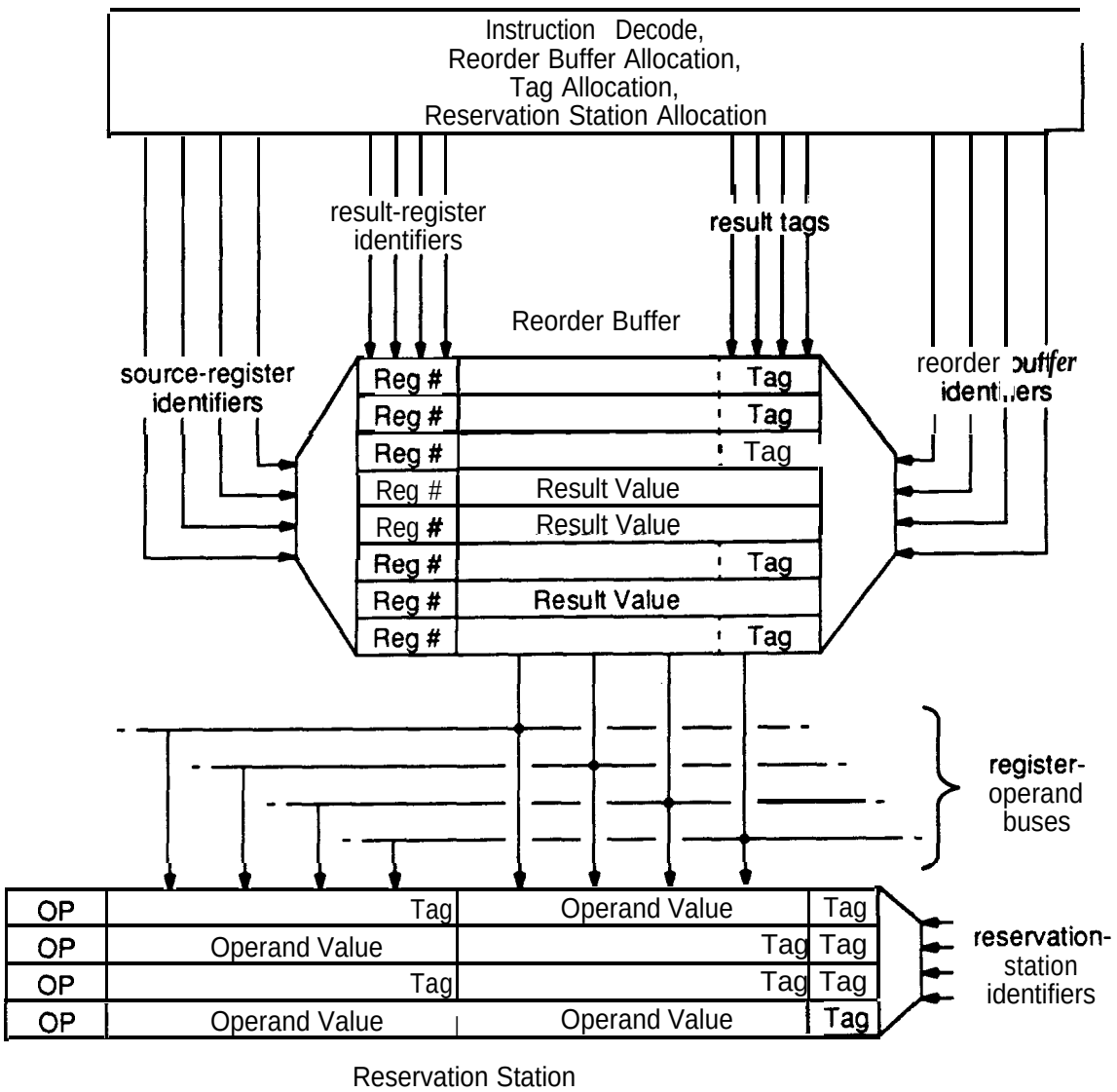

Figure 50. Instruction Decode Stage

for distribution. This minor optimization does not handle all requirements for tag distribution, as discussed below. If instructions arbitrate for operand buses, as assumed here, the reservation stations must match instructions to operands or tags, so that these can be stored properly.

Supplying operands to reservation stations requires two paths that are not shown in Figure 50 (these are omitted for clarity). The first of these paths distributes instruction-immediate data from the decoder to the reservation stations. The second path distributes tags for instructions that depend on instructions which are decoded in the same cycle. These tags

are supplied by the tag-allocation hardware, rather than the reorder buffer. If the instruction format explicitly identifies instruction dependencies, such as that described in Section 4.4.2, the decoder can easily distribute the appropriate tags. Otherwise, this must be based on comparisons between destination-register and source-register identifiers of decoded instructions, taking twelve comparators. In either case, the dependent operands are not allocated operand buses, requiring other buses to distribute tags to the reservation stations.

There is a potential pipeline hazard in the forwarding logic [Weiss and Smith 1984] that the design must avoid. This hazard arises when a decoded instruction obtains a result tag at about the same time that the result is written. In this case, it is possible that the forwarding logic obtains a result tag, but misses the write-back of the corresponding result, causing incorrect operation. Avoiding this hazard may may take eight additional comparators to detect, during decode, that forwarding is required in the next cycle.

At the end of the decode cycle, the result-register identifiers and result tags for decoded instructions are written into the reorder buffer. These should be written into the empty locations nearest the head of the buffer, as identified by the allocation hardware. This requires four write ports on the storage for register identifiers and tags at each location of the reorder buffer. The four write ports could be avoided by allocating entries only at the tail of the reorder buffer, in which case only these entries would require four write ports. However, allocating entries at the tail of the reorder buffer complicates the associative lookup, because it becomes difficult to determine the location of valid entries. Furthermore, this approach increases the amount of time taken for a result to reach the head of the reorder buffer, increasing the branch delay if the entire reorder buffer is flushed on a mispredicted branch as discussed in Section 5.2.1.

With respect to updating the dependency logic, renaming and partial renaming with a future file also require four ports for writing tags into a tag array. The reorder buffer has the advantage that it is generally smaller than the register file, and takes fewer write decoders. In the example described here, the reorder buffer has half the number of entries of the register file.

5.6.3 Instruction Completion

When a functional unit generates an instruction result, the result is written into the reorder buffer after the reservation station has successfully arbitrated for the result bus. The result tag must be used to identify which reorder-buffer entry is to be written. The reorder-buffer identifier that was allocated during decode was used only to write the result tag and the result-register identifier into the reorder buffer, and is meaningless in subsequent cycles

because entries advance through the reorder buffer as results are written to the register file. Thus, this write must be associative: the written entry is the one with a result tag that matches the result tag of the completed instruction. This is true of any ***reorder*** buffer operated as a FIFO, with or without a future file.

5.7 Observations and Conclusions

Instruction restart is an integral part of the super-scalar processor, because it allows instruction decoding to proceed at a high rate. Even the software approaches to improving the fetch efficiency proposed in Section 4.2.4 require nullifying the effect of a variable number of instructions. Furthermore, there is little sense in designing a super-scalar processor with in-order completion, because of the detrimental effect on instruction-level concurrency. Restarting a processor with out-of-order completion requires at least some of the hardware described in this chapter. If out-of-order issue is implemented, the reorder buffer provides a very useful means for recovering the instruction sequentiality that is given up after decode; for example, it helps in committing stores to the data cache at the proper times. Finally, the buffering provided for restart, with associative lookup, helps manage the look-ahead state and simplifies operand dependency analysis.

The best implementation of hardware restart has either a twelve-entry or a sixteen-entry reorder buffer, with every instruction being allocated an entry in this buffer. A future file provides a way to avoid associative lookup in the reorder buffer, at the prohibitive cost of an additional register array. The future file also adds delay penalties to mispredicted branches, because it requires that the processor state be serialized up to the point of the branch before instruction execution resumes.

If software alone cannot insure the absence of dependencies, renaming with a reorder buffer is the best hardware alternative for resolving operand dependencies. Unfortunately, dependency analysis is fundamentally hard, especially when multiple instructions are involved. All of the alternatives to renaming explored in this chapter require a comparable or greater amount of hardware, with reduced performance. The example implementation of renaming presented in this chapter takes: 64, five-bit comparators in reorder buffer for reading operands; 32, four-bit comparators in reorder buffer for writing results; 60, four-bit comparators in reservation stations for forwarding; logic for allocating result tags and reorder-buffer entries; and a four-read-port, two-write-port reorder buffer that has four additional write ports on portions of the entries for writing register identifiers, results tags, and instruction state. The complexity of this hardware is set by the number of uncompleted instructions

permitted, the width of the decoder, the requirement to restart after a **mispredicted** branch, and the requirement to forward results to waiting instructions.

The associative logic and multiple-port arrays are unattractive, but are not easily eliminated. Without the associative logic for forwarding, the super-scalar processor has worse performance than a scalar processor. Adding a future file to the architecture can eliminate the associative read in the reorder buffer, but a future file also requires tag comparisons in the **write**-back logic to implement renaming and incurs the cost of the future file itself. The tag comparisons in the write-back logic can be eliminated by partial renaming (which updates registers in order), but the logic to guarantee that registers are updated in order is more complex than the eliminated logic. Thorton's algorithm can be used with a future file to eliminate the associative lookup in the reorder buffer, and does not require tag-comparison logic to write results into the future file. However, Thor-ton's algorithm yields the poorest performance of any alternative explored in this chapter, and requires a technique for detecting during decode that result registers have pending updates. Considering also the cost of the four-write-port, two-read-port tag array for the future file, any approach using a future file holds little promise.

The cost of the dependency and forwarding hardware should be viewed as a more-or-less fixed cost of attaining a high instruction-execution rate. Of course, this can lead to the conclusion that such an instruction-execution rate is too ambitious. The argument offered here is only that there are no simple hardware alternatives yet discovered that yield correct operation with performance better than in-order issue. The following chapter considers other ways to reduce the amount of hardware required, by focusing on alternatives to reservation stations.

Chapter 6
Instruction Scheduling and Issuing

The peak instruction-execution rate of a super-scalar processor can be almost twice the average instruction-execution rate, as shown in Figure 6 (page 27). This is in contrast to a scalar **RISC processor,** which typically has an average execution rate that is within 30% of the peak rate. A straightforward approach to super-scalar processor design is to simply provide enough hardware to support the peak instruction-execution rate. However, this hardware is expensive, and the difference between the average and peak execution rates implies that the hardware is under-utilized.

A more cost-effective approach to processor design is to carefully constrain the hardware in light of the average instruction-execution rate. Several examples of this approach have been seen in preceding chapters. Chapter 4 demonstrated that the number of **register-file** read ports could be constrained to **four**—approximately the number of ports needed to sustain an execution rate of two instruction per cycle-even with a four-instruction decoder. Similarly, Chapter 5 demonstrated that the tag logic associated with the register file in Tomasulo's algorithm can be consolidated into a reorder buffer that requires less tag hardware than the register file because it tracks only pending register updates.

The reservation stations described in Table 3 (page 32) are also under-utilized. These reservation stations-used to obtain all results so far-have a total capacity of thirty-four instructions. However, Chapter 5 showed that the reorder buffer should have twelve to sixteen locations. Since the total allocation for the reorder buffer is determined by instructions in the window ***and*** in the pipeline stages of the functional units, this indicates that the instruction window contains fewer than about twelve instructions.

The problem of under-utilization of the reservation stations is overcome by consolidating them into a smaller ***central*** window. The central window holds all unissued instructions, regardless of the functional units required by the instructions, as illustrated by Figure 51 (only the integer functional units are shown in Figure 51, and the central window may be duplicated in the floating-point unit, depending on the implementation). For a given level of performance, the central window uses much less storage than reservation stations, less control hardware, and less logic for resolving dependencies and forwarding results.

This chapter presents the motivations for the reservation-station sizes given in Table 3, then examines two published central-window proposals in an attempt to reduce hardware

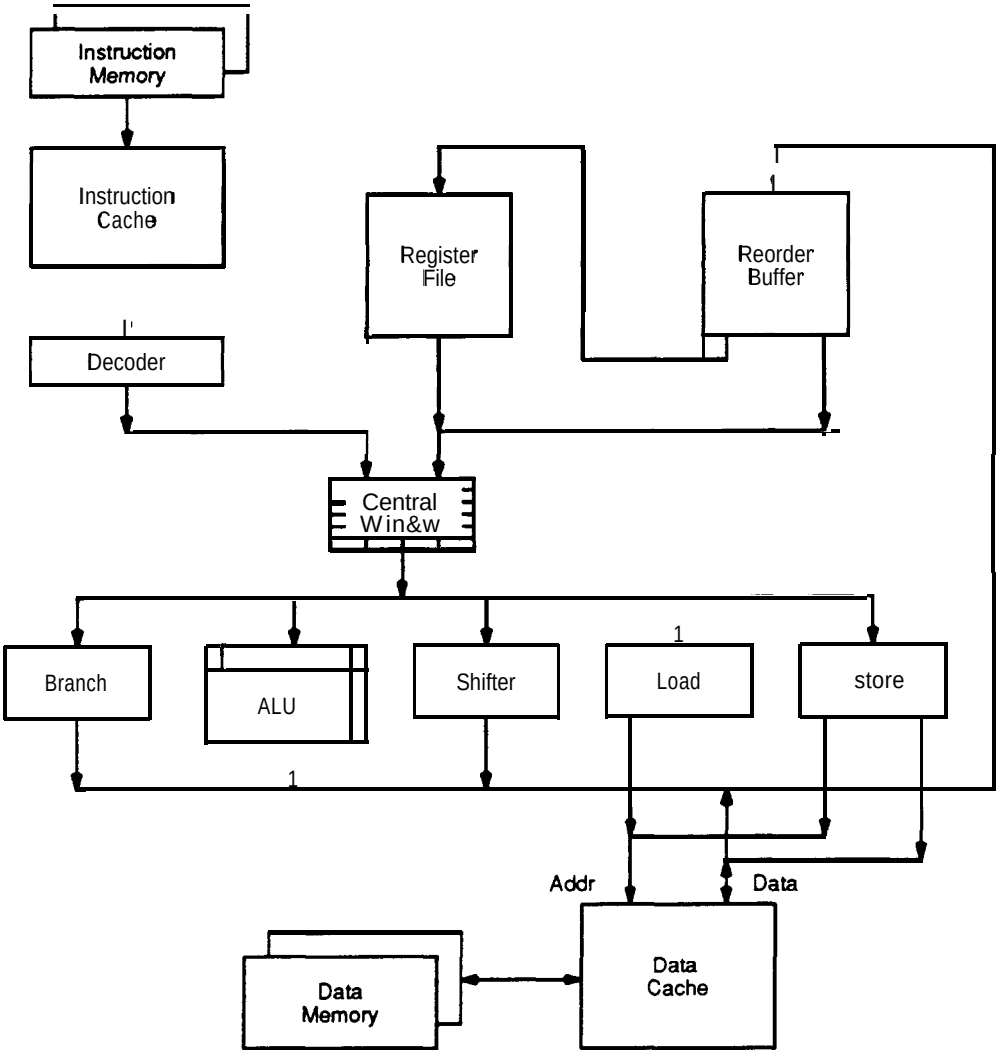

Figure 51. Location of Central Window in Processor (Integer Unit)

requirements: ***the dispatch stuck*** proposed by Tomg [1984] and the ***register update unit*** of Sohi and Vajapeyam [1987]. Both of these proposals have disadvantages: the dispatch stack is complex, and the register update unit has relatively poor performance. In both proposals, the disadvantages are the result of using the central window to implement operations that are better implemented by the reorder buffer. A third proposal presented in this chapter relies on the existence of the reorder buffer and associated dependency logic to simplify the central window. This serves as an additional illustration of the synergy between hardware components in the super-scalar processor.

The second major topic of this chapter is the implementation of loads and stores with a central instruction window. This chapter demonstrates how load and store hardware can be simplified, in a processor that uses a central window to issue instructions. However, it should be noted that the benchmark programs used in this study do not have a high frequency of loads and stores compared to many scientific applications, and thus the performance of these benchmark programs is not very sensitive to limitations in the data-memory interface.

6.1 Reservation Stations

Reservation stations partition the instruction window by functional unit, as shown in Figure 7 (page 29). This partition helps simplify to control logic at each reservation station, but causes total number of reservation-station entries to be much larger than required for a central instruction window.

6.1.1 Reservation Station Operation

A reservation-station entry holds a decoded instruction until it is free of dependencies and the associated functional unit is free to execute the instruction. The following steps are taken to issue instructions from a reservation station:

1) Identify entries containing instructions ready for issue. An instruction is ready when it is the result of a valid decode cycle, and all operands are valid. Operands may have been valid during decode, or may have just become valid because a result has been computed.

2) If more than one instruction is ready for issue, select an instruction for issue among the ready instructions.

3) Issue the selected instruction to the functional unit.

4) Deallocate the reservation-station entry containing the issued instruction, so that this entry may receive a new instruction. For best utilization of the reservation station, the entry should be able to receive an instruction from the decoder in the next cycle.

These functions are performed in any instruction window that implements out-of-order issue. Reservation stations have some advantage, though, in that these functions are partitioned and distributed in a way that simplifies the control logic. Most reservation stations perform these functions on a small number of instructions. The load and store reservation stations are relatively large (eight entries each), but these reservation stations are constrained to issue instructions in order, and so there is only one instruction that can be

issued-the issue logic simply waits until this instruction is ready rather than selecting among ready instructions. Furthermore, a reservation station can free at most one instruction per cycle, so deallocation is not very difficult. Of course, reservation stations require multiple instances of the issuing and allocation logic.

6.1.2 Performance Effects of Reservation-Station Size

Instructions are supplied to the reservation stations by an instruction decoder, and instruction decoding stalls if a required functional unit has a full reservation station. Reservation station entries become full if either local data dependencies or functional-unit conflicts cannot be resolved by the execution unit at a sufficient rate compared to the rate of instruction fetching anddecoding. The reservation stations must be large enough to handle the majority of short-term situations where the instruction-fetch rate exceeds the instruction-execution rate. Otherwise, the mismatch in rates stalls the decoder and prevents the decoding of instructions for other functional units, reducing the instruction bandwidth.

Figure 52 shows the effect of reservation-station size on the performance of the super-scalar processor. To simplify the presentation of data, these results were measured with all reservation stations having the same size. With a two-instruction decoder, performance is not reduced significantly until the reservation stations are reduced to two entries each. With

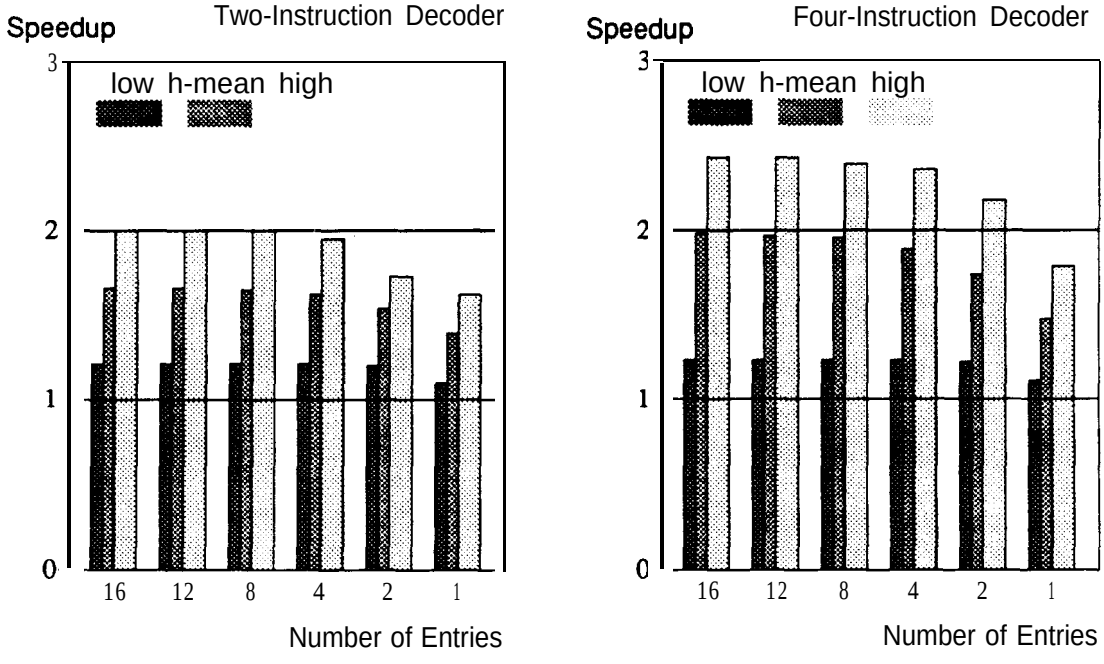

Figure 52. Performance Effect of Reservation-Station Size

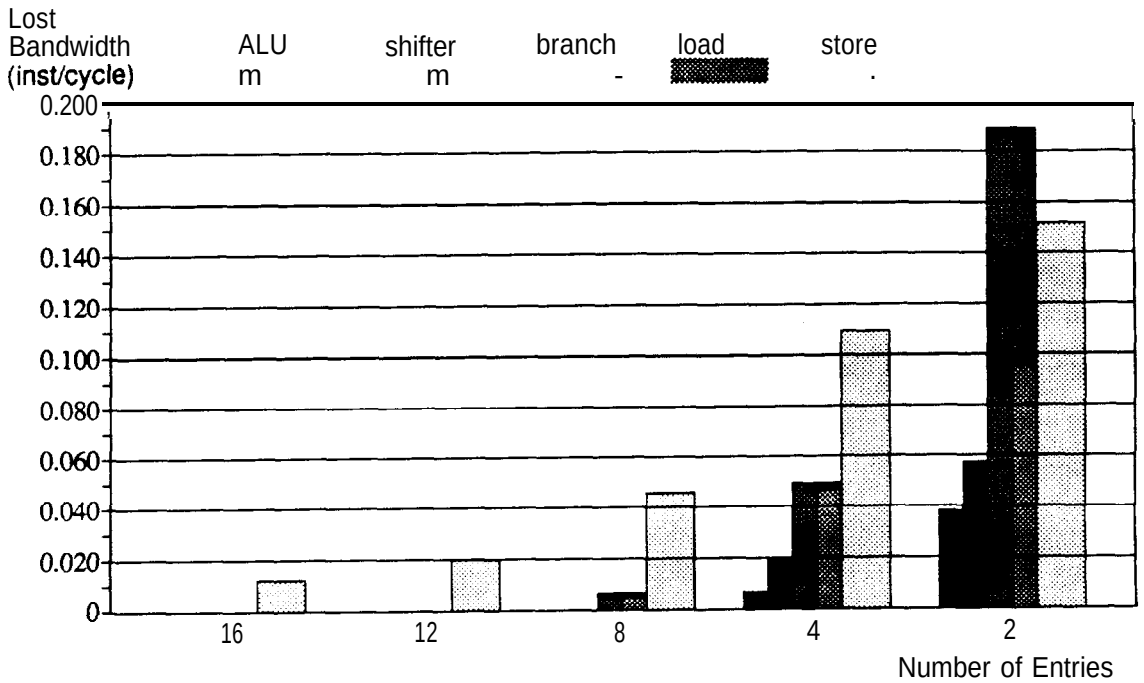

Figure 53. Effect of Reservation Station Size ***on the Average Instruction-Fetch Bandwidth Lost Because of a Full Reservation Station, Per Integer Functional Unit***

more reservation-station entries, a processor with a two-instruction decoder is instruction-fetch limited. With a four-instruction decoder, performance is noticeably reduced when reservation stations are reduced to four entries each. In reality, however, not all reservation stations require four entries. Experience with the simulator indicates that the reservation-stations sizes given by Table 3 provide good performance.

Figure 53 illustrates, for the sample benchmarks, the role that the reservation stations serve in providing adequate instruction bandwidth (by preventing decoder stalls). Figure 53 shows, by integer functional unit, the average instruction-fetch bandwidth lost at the decoder because of full reservation stations, as the sizes of the reservation stations are reduced (the floating-point reservation stations caused negligible penalties for these benchmarks). Reservation-station size is most critical for the branch, load, and store functional units. These functional units are constrained, for correctness, to execute instructions in their original order with respect to other instructions of the same type, and thus have the highest probability of a mismatch between instruction-fetch and instruction-execution rates. The store functional unit is generally the worst in this respect: stores are constrained to issue only after all previous instructions have completed, then contend with loads (with low priority) for use of the address bus to the data cache. However, a small branch reservation station causes the

most severe instruction-bandwidth limitation, because branches are more frequent than stores and because branches are likely to depend on the results of a series of instructions (this dependency is the reason that the branch delay is typically three or four cycles).

Reservation stations are under-utilized because they serve two purposes: they implement the instruction window (requiring buffering for less than sixteen instructions), and they prevent local demands on a functional unit from stalling the decoder (taking two to eight entries at each functional unit, for a total of thirty-four entries). The latter consideration is most important for the branch, load, and store functional units, but causes the total number of reservation-station entries for all functional units to be higher than the number ***required*** solely for the instruction window.

6.2 Central Instruction Window

As stated previously, a central window is a more efficient implementation of the instruction window than reservation stations, because it holds all instructions for issue regardless of functional-unit requirements. The central window consolidates all instruction-window entries and scheduling logic into a single structure. However, the scheduling logic associated with a central window is more complex than the corresponding logic in a reservation station, for several reasons:

- The central window selects among a larger number of instructions than does a reservation station. The central window schedules all instructions in the window, rather than a subset of these instructions as is the case with a reservation station.
- The central window **must** consider functional-unit requirements in selecting among ready instructions.
- The central window can free more than one entry per cycle, so allocation and deallocation are more complex than with reservation stations.

This section explores two implementations of a central window-the dispatch stack [Tomg 1984] and the register update unit [Sohi and Vajapeyam 1987]—and proposes a third implementation that has much better performance than the register update unit without the complexity of the dispatch stack. The proposed implementation relies on the dependency logic described in Chapter 5, and thus relies on the existence of a reorder buffer, whereas the dispatch stack and register update unit incorporate some of the function of the reorder buffer. However, the reorder buffer is an important component for supporting branch prediction and

register renaming: the fact that the reorder buffer can simplify the central window is a further advantage of the reorder buffer.

It is also shown in this section that the number of buses requited to distribute operands from a central window is comparable to the number of buses required to distribute operands to reservation stations-negating the concerns expressed in Section 3.3.1. The central window can schedule instruction issue around the operand-bus limitation, illustrating that the instruction window allows execution hardware to be constrained without much reduction in performance. However, arbitrating for operand buses further complicates the scheduling logic associated with the central window.

6.2.1 The Dispatch Stack

In every cycle, the dispatch stack [Tomg 1984, Acosta et al. 1986, Dwyer and Tomg 1987] performs operations that are very similar to those performed in a reservation station:

1) Identify entries containing instructions ready for issue.

2) Select instructions for issue among the ready instructions, based on functional-unit requirements. This involves prioritizing among instructions that require the same functional unit.

3) Issue the selected instructions to the appropriate functional units.

4) Deallocate the reservation-station entries containing the issued instructions, so that these entries may receive new instructions.

The dispatch stack keeps instructions in a queue sorted by order of age, to simplify dependency analysis (dependencies are resolved in the dispatch-stack proposal by usage and definition counts, as described in Section 2.3.4) and to simplify prioritizing among ready instructions (it is easy to give priority to older instructions because these are nearer the front of the stack). To keep instructions in order, the dispatch stack cannot just allocate entries by placing decoded instructions into freed locations: it must fill these locations with older, waiting instructions and allocate entries for new instructions at the end of the stack, as shown in Figure 54.

Figure 55 shows the effect of dispatch-stack size on the performance of the super-scalar processor. This performance was measured with a dispatch stack that can accept a portion of the decoded instructions when the stack does not have enough entries for all decoded instructions. Also, an instruction can remain in the dispatch stack for an arbitrary amount of time. Both a two-instruction and a four-instruction decoder decoder require only an eight-

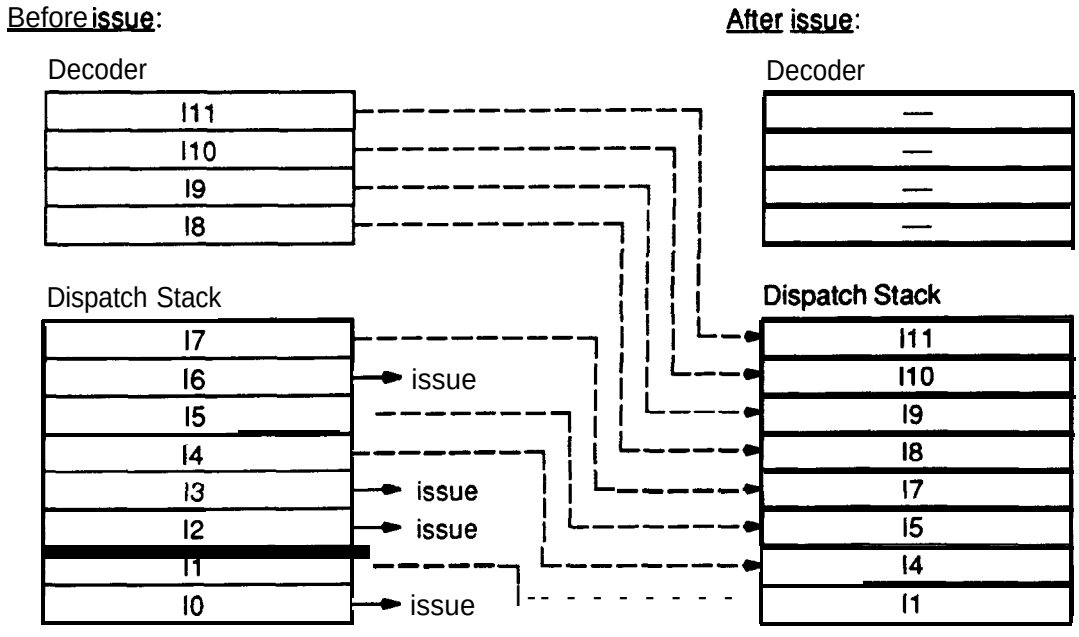

Figure 54. Compressing the Dispatch Stack

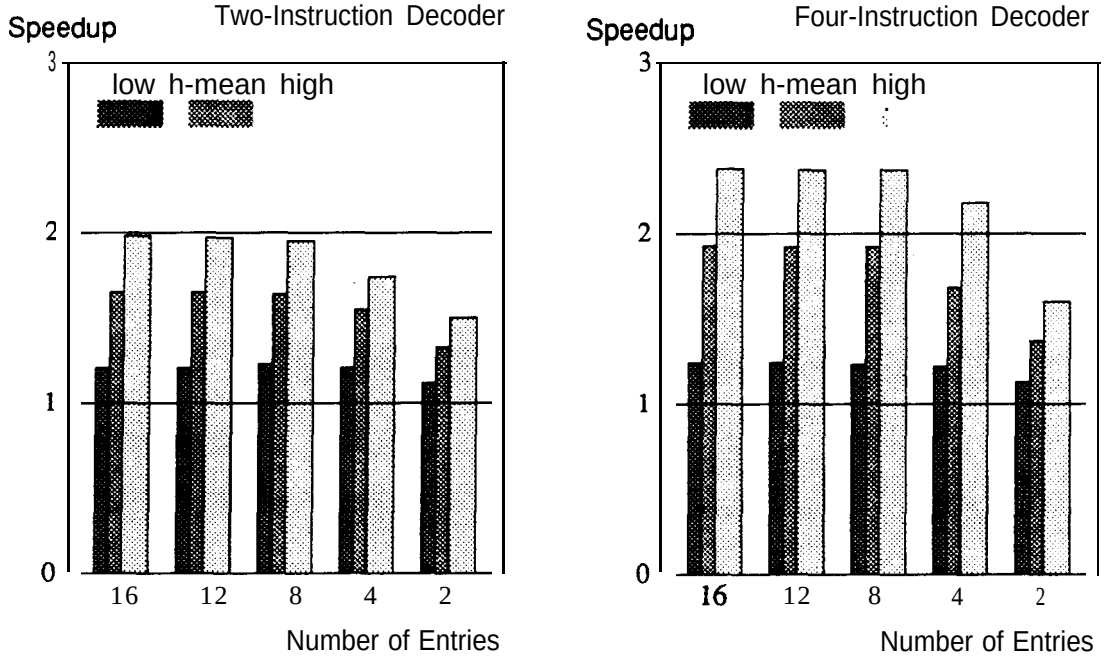

Figure 55. Performance Effect of Dispatch-Stack Size

entry dispatch stack for best performance. This is consistent with the size expected from the size of the reorder buffer. Thus, with a four-instruction decoder, the dispatch stack uses about one-fourth of the storage and forwarding logic of the reservation stations. Furthermore, the instruction-scheduling logic is not duplicated (unless it is duplicated in separate integer and floating-point units). With the reservation stations in Table 3, the instruction-issue logic is replicated ten times.

Unfortunately, instruction scheduling with a dispatch stack is complex, because the dispatch stack keeps instructions in order. Compressing and allocating the window can occur only after the window has identified the number and location of the instructions for issue. Compressing the window also requires that any entry be able to receive the contents of any subsequent (newer) entry and that the entries for the newest instructions be able to receive the instruction from any decoder slot. Thus, for example, an entry in an eight-location window with a four-instruction decoder requires a maximum of eleven possible input sources, a minimum of four, and an average of about eight. Dwyer and Tomg [1987] estimate that the issue, compression, and allocation logic for a proposed implementation of an eight-entry central window consume 30,000 gates and 150,000 transistors.

6.2.2 The Register Update Unit

Although a dispatch stack requires less storage for instructions and operands than do reservation stations, the complexity of the issue, compression, and allocation hardware in a dispatch stack argues against its use. The register update unit [Sohi and Vajapeyam 1987] is a simpler implementation of a central window that avoids the complexity of compressing the window before allocation.

The operations performed to issue instructions in the register update unit are identical to those in the dispatch stack (the criteria for issue are different, though, because the dependency mechanism is different). However, the register update unit allocates and frees window entries in a first-in, first-out (FIFO) manner, as shown in Figure 56. Instructions are entered at the tail of the FIFO, and window entries are kept in sequential order. An entry is not removed when the associated instruction is issued. Rather, an entry is removed when it reaches the head of the FIFO, When an entry is removed, its result (if applicable) is written to the register file. This method was originally proposed to aid the implementation of precise interrupts, but it also decouples instruction issuing from window allocation and uses a simple algorithm to allocate window entries.

Figure 57 shows the performance of the register update unit, as a function of size. The set of bars in the foreground of Figure 57 show the speedup obtained with a register update unit,

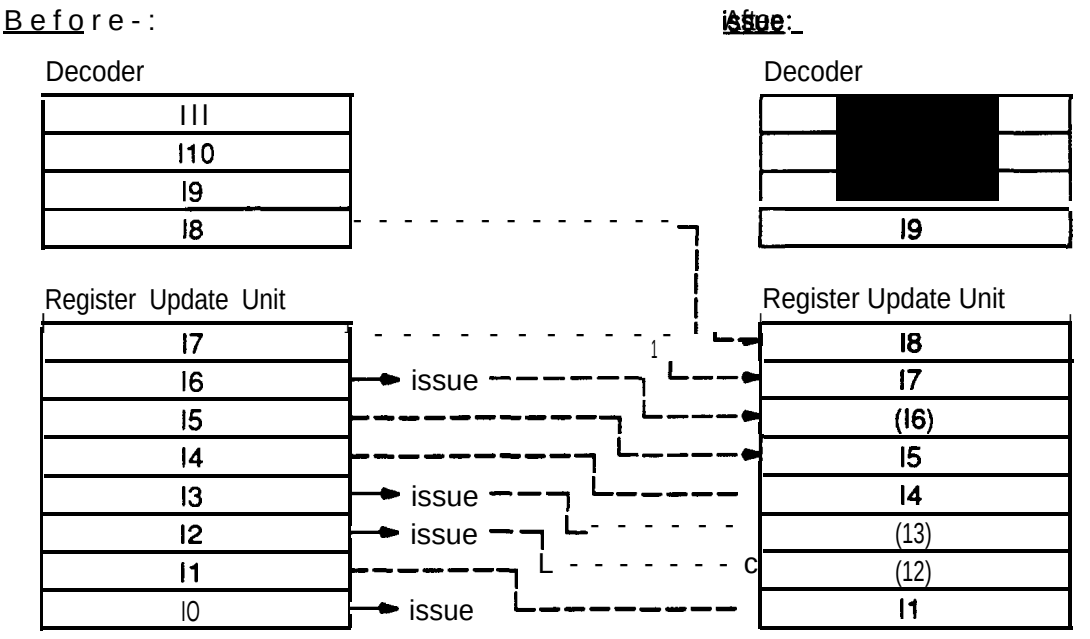

Figure 56. Register Update Unit Managed as a FIFO

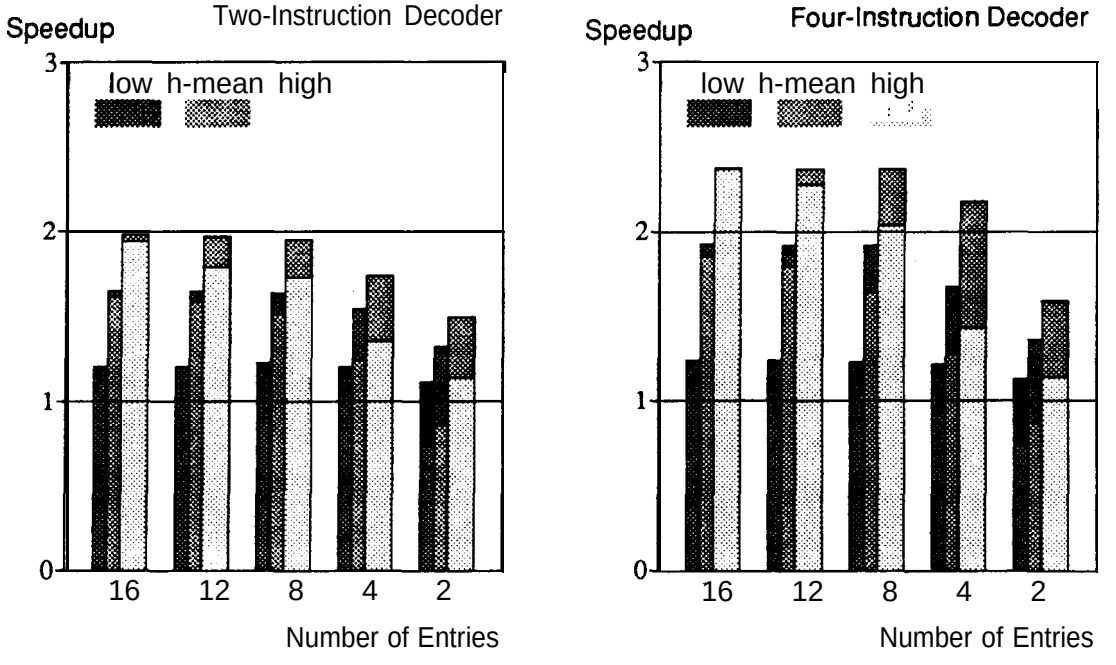

Figure 57. Performance Degradation of Register Update Unit Compared to Dispatch Stack

and the set of bars in the background repeat the data for the dispatch stack, from Figure 55, for comparison. For all sizes shown, performance of the register update unit is significantly lower than the performance of a dispatch stack of equal size. The performance disadvantage is most pronounced for smaller window sizes (the reason performance can be lower than that of the scalar processor is that instructions remain in the register update unit for at least one cycle before being written to the register file, possibly stalling the decoder for longer periods than if results are written directly to the register file). Sohi and Vajapeyam [1987] report similar results-they find that a fifty-entry register update unit provides the best performance, in a processor that has higher functional-unit latencies that the processor used in the current study.

There are two reasons that the register update unit has this performance disadvantage (Figure 56 helps to illustrate these points):

- Window entries are used for instructions which have been issued.
- Instructions cannot remain in the window for an arbitrary amount of time without reducing performance. Rather, the decoder stalls if an instruction has not been issued when it reaches the head of the register update unit.

Figure 58 shows, for the dispatch stack described in Section 6.2.1, the average distribution of the number of cycles from instruction decode to instruction issue. With the register update unit, instructions at the high end of this distribution stall the decoder, because they cannot be issued and complete by the time they arrive at the front of the window. As discussedin Section 6.1.2, stores experience the worst decode-to-issue delay, because they can be issued in order only after all previous instructions are complete and then contend with loads for the data cache.

It should be emphasized that Sohi and Vajapeyam [1987] proposed the register update unit as a mechanism for implementing precise interrupts and restarting after mispredicted branches. The disadvantages cited here are due to the different objective of reducing the size of the instruction window.

6.2.3 Using the Reorder Buffer to Simplify the Central Window

Both the complexity of the dispatch stack and the relatively low performance of the register update unit can be traced to a common cause: both proposals attempt to maintain instructions ordered in the central window. However, a reorder buffer is much better suited to maintaining instruction-ordering information. The reorder buffer is not compressed as is the

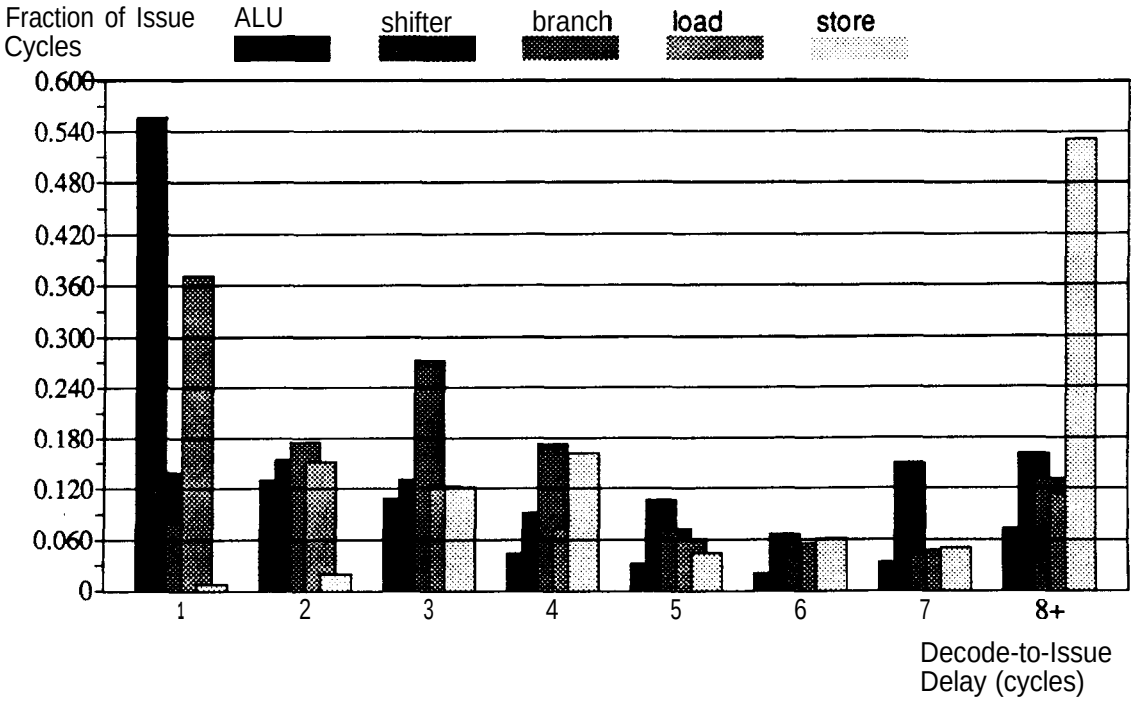

Figure 58. Distribution of Decode-to-Issue Delay for Various Functional Units

dispatch stack, and the deallocation of reorder-buffer entries is not in the critical path of instruction issue. Rather, reorder-buffer entries are allocated at the tail of a FIFO and deallocation occurs as results are written to the register file. Also, in contrast to the register update unit, instructions will not stall the instruction decoder if they are not issued by the time the corresponding entry reaches the head of the reorder buffer. As described in Section 5.2.2, for example, stores are simply released for issue at the output of the reorder buffer. Several stores can be released in a single cycle, so the reorder buffer does not suffer the single-issue constraints of a store in the register update unit. Consequently, the reorder buffer does not stall instruction decoding as readily as the register update unit. The different method for handling stores is the primary advantage of this approach over the register update unit.

In an implementation with a reorder buffer, then, there is no real need to keep instructions ordered in the central window. The window is not compressed as instructions are issued. Instead, new instructions from the decoder are simply placed into the freed locations, as Figure 59 shows. The principal disadvantage of this approach is that there is no instruction-ordering information in the instruction window, and it is not possible to prioritize instructions for issue based on this order. In general, when two instructions are ready for issue at the same time and require the same functional unit, it is better to issue the older instruction,

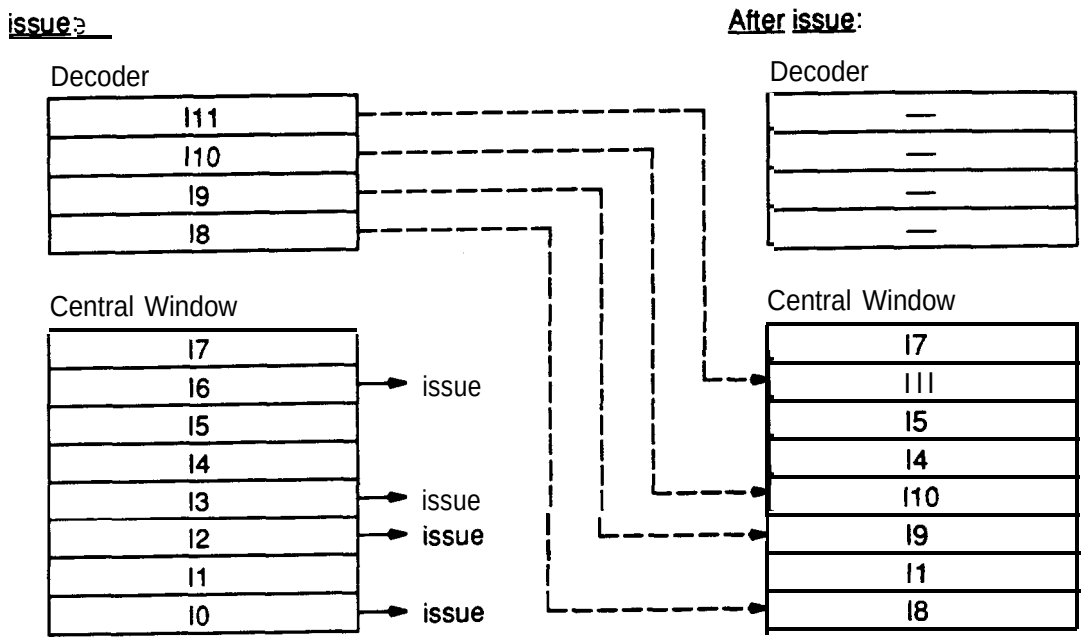

Figure 59. Allocating Window Locations without Compressing

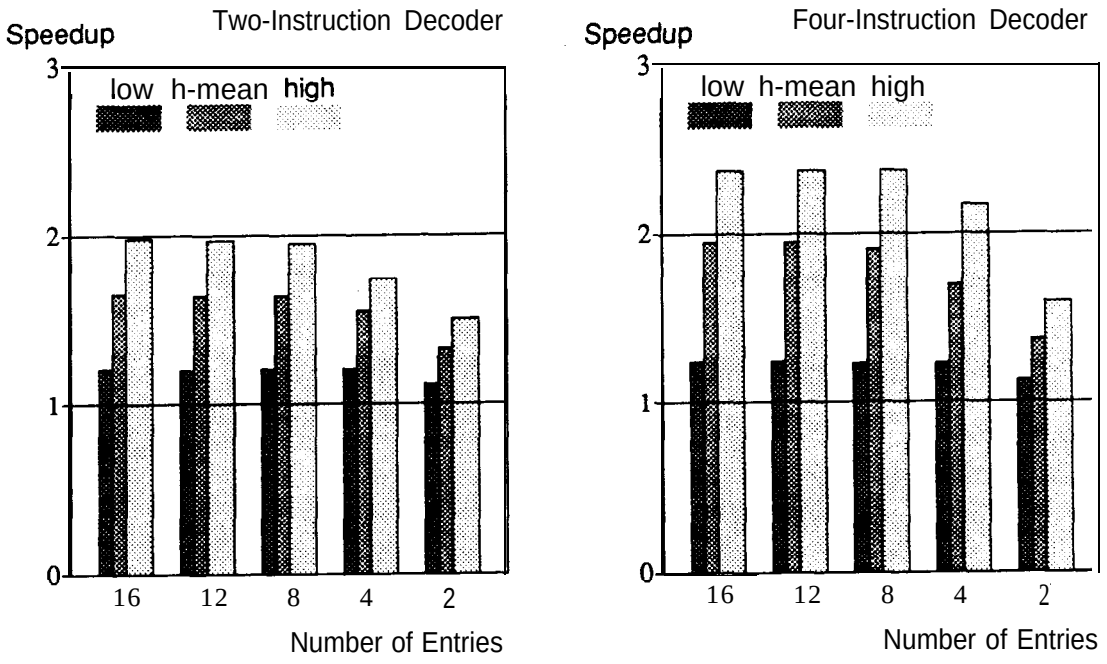

Figure 60. Performance Effect of Central-Window Size without Compression

because this is more likely to free other instructions for issue. Still, the data in Figure 60 suggest that this is not a very important consideration. Figure 60 gives performance as function of central-window size when the window is allocated without compression as shown in Figure 59. In this configuration, instructions are prioritized for issue based on their position in the central window, but this position has nothing to do with the original program order. Compared to the performance of the dispatch stack, the reduction in performance caused by this less-optimal prioritization is less than 1% for any benchmark-the reduction is less than the measurement precision for most benchmarks. Obviously, the dependencies between instructions are sufficient to prioritize instruction issue.

Even though instructions do not have to be ordered in the window for performance, some instructions (loads, stores, and branches) must be issued in original program order. The central window proposed here does not preserve instruction ordering, and thus complicates issuing instruction in order. This difficulty is easily overcome by allocating sequencing tags to these instructions during decode if there is another, unissued instruction of the same type in the window. When a sequenced instruction is issued, it transmits its tag to all window entries, and the window entry with a matching sequencing tag is released for issue.

6.2.4 Operand Buses

A concern with a central window is the amount of global (chip-wide) interconnection required to supply operands to the functional units from the central window. The process of distributing instructions and operands is significantly different for a central window than it is for reservations stations. Instructions and operands are placed into the central window using local interconnections from the decoder, reorder buffer, and register file. The window in turn issues these instructions and operands to the functional units using global interconnections. In contrast, instructions and operands are placed into reservation stations using global interconnections. The reservation stations in turn issue these instructions and operands to the functional units using local interconnections.

Section 4.4 established that a limit of four register-based operands is sufficient to supply reservation stations, and does not significantly reduce performance. This gives an estimate of the number of interconnection buses required for the distribution of instructions and operands to reservations stations. However, this number of operands is sufficient because empty decoder slots and dependencies between decoded instructions reduce the demand on register-based operands during decode. If instructions and operands are issued to the functional

units from a central window, these empty slots and dependencies do not exist, because the instructions are being issued for execution.

Interconnection networks are much less dense than storage elements, so the interconnect required by a central window might consume more chip area than the additional storage of the reservation stations. Because of this consideration, Figure 61 shows the performance of a central window of sixteen entries for various numbers of operand buses. A large, sixteen-entry window is used in order to focus on the effect of limited interconnection. As shown, the number of operand buses can be limited to four (equal to the number of buses adequate to distribute operands to reservation stations) with negligible effect on performance.

The reason that limiting the number of buses has so little effect is that the instruction window is able to schedule instruction issue around this limitation. Figure 62 illustrates this point by comparing the average instruction-issue distribution from a central window with eight operand buses to the same distribution with four operand buses. As shown, limiting the number of buses reduces the percentage of cycles in which a high number of instructions are issued, but the instructions not issued are likely to be issued later with other instructions, and the average execution rate is maintained.

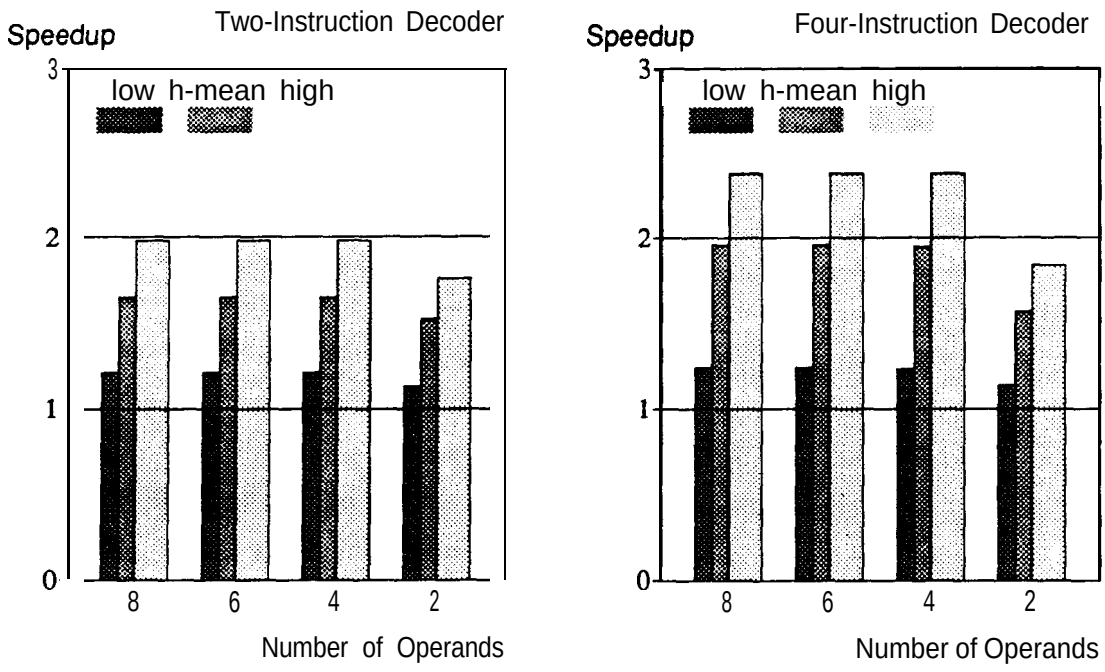

Figure 61. Performance Effect of Limiting Operand Buses from a Sixteen-Entry Central Window

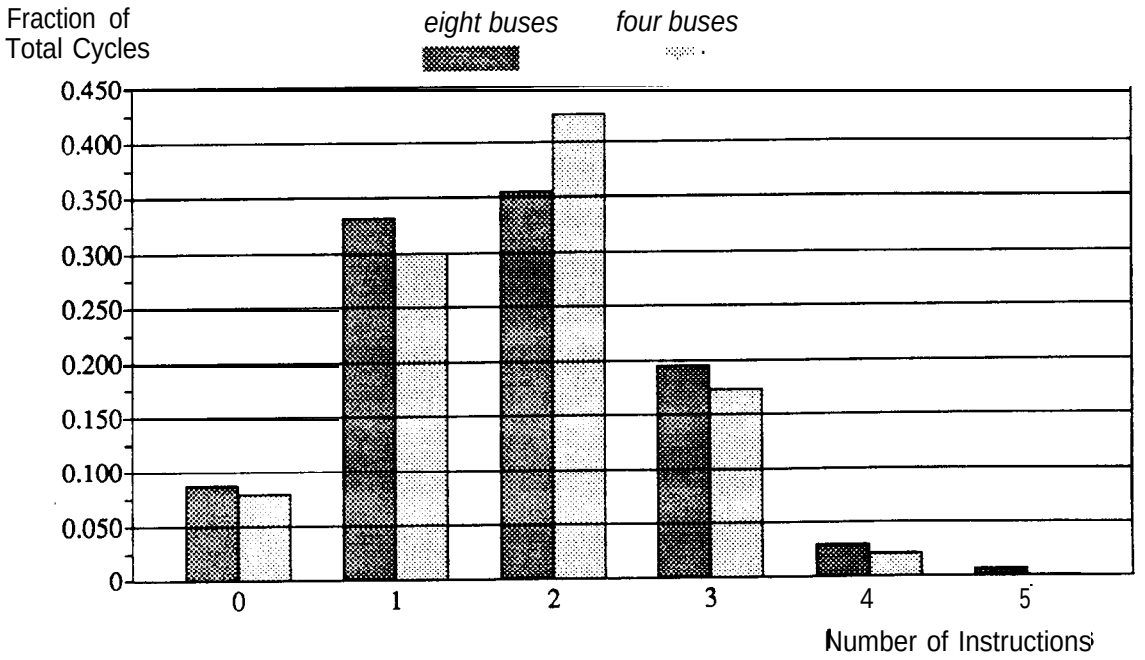

Figure 62. Change in Average Instruction-Issue Distribution from Sixteen-Entry Central Window as Operand Buses are Limited

6.2.5 Central Window Implementation

The primary difficulty of operand distribution with the central window is not that too many buses are required, but the fact that operand-bus arbitration must be considered during instruction scheduling. The arbitration for operand buses is conceptually similar to the register-port arbiter discussed in Section 4.4.1, except that there are as many as sixteen operands contending for the four shared buses. Furthermore, it is more difficult to determine which of the contenders are actually able to use the buses. For example, to determine which instructions are allowed to use operand buses, it is first necessary to determine which instructions are ready to be issued and which functional units are ready to accept new instructions. Following this, functional-unit conflicts among ready instructions must be resolved before instructions arbitrate for operand buses. Fortunately, there is somewhat more time to accomplish these operations than there is with register-port arbitration, because register-point arbitration typically must be completed by the mid-point of a processor cycle.

The longest identifiable path in the instruction-scheduling logic arises when an instruction is about to be made ready by a result that will be forwarded. In this case, the result tag is valid the cycle before the result is valid, and a tag comparison in the instruction window readies the instruction for issue. Once the scheduling logic determines that the instruction is ready,

the instruction participates in functional-unit arbitration, then operand-bus arbitration. If the instruction can be issued, the location it occupies is made available for allocation to a decoded instruction. Table 5 gives an estimate of the number of stages required for each of these operations, using serial arbitration as described in Section 5.6.1 for the complex arbiters and assuming no special circuit structures (such as wired logic). The sixteen stages required are within the capabilities of most technologies.

It is interesting to note that four of the operations in Table 5 are concerned with resource arbitration and allocation. That these operations have also been examined within the contexts of instruction fetching and dependency analysis suggests that hardware structures for arbitration and allocation are important areas for detailed investigation. Sustaining an instruction throughput of more than one instruction per cycle with a reasonably small amount of hardware creates many situations where multiple operations are contending for multiple, shared resources. The speed and complexity of the hardware for resolving resource contention is easily as important as any other implementation consideration addressed in this study, but it is impossible to determine the feasibility of this hardware without knowing precisely the details of the implementation.

Despite its additional complexity, the central window is preferred over reservation stations. It uses many fewer storage locations and comparators for forwarding than do reservation stations, and requires comparable busing to distribute operands.

6.3 Loads and Stores

Generally, it is not possible to improve concurrency of loads and stores by issuing them out-of-order, except that loads can be issued out-of-order with respect to stores if each load

Table 5. Critical Path for Central-Window Scheduling

Function in Path	Number of Stages
result-bus arbitration	2
drive tag buses	1
compare results tags to operand tags and determine ready instructions	3
functional-unit arbitration	2
operand-bus arbitration	4
allocate free window entries to decoded instructions	4
TOTAL:	16

checks for dependencies on pending stores. As already discussed, stores are serialized with all other instructions to preserve a sequential memory state for restart This section examines alternative mechanisms for access ordering, address computation and translation, and memory dependency checking. Reservation stations provide much leeway in the load and store mechanisms. For this reason, this section begins by considering implementations using reservation stations, to illustrate general implementations. However, since the conclusion of Section 6.2 is that a central instruction window is preferred over reservation stations, this section develops an implementation of loads and stores using a central instruction window.

Throughout this section, the timing and pipelining characteristics of the address computation and translation unit and of the data-cache interface are assumed to be comparable to those of the MIPS R2000 processor. This avoids distorting the results with effects that are not related to the super-scalar processor, but is not meant to imply that these are the only approaches to implementing these functions. For example, the overhead involved in issuing a load or store in the super-scalar processor may prevent computing and translating an address in a single cycle (these operations are performed in a single cycle in the R2000 processor). The additional overhead may motivate another approach to address translation, such accessing the data cache with virtual addresses rather than physical addresses. However, such considerations are not explicitly examined here.

This section considers ways of relaxing issue constraints on loads and stores while maintaining correct operation. Throughout this section, the following code sequence is used to illustrate the operation of the various alternatives:

```
STORE v      (1)
ADD          (2)
LOAD  w      (3)
LOAD  x      (4)
LOAD  v      (5)
ADD          (6)
STORE w      (7)
```

In this instruction sequence, the `LOAD` v instruction (line 5) depends on the value stored by the `STORE` v instruction (line 1). Technically, the `STORE` w instruction (line 7) anti-depends on the instruction `LOAD` w (line 3), but, because stores are held until all previous instructions complete, anti-dependencies on memory locations are not relevant to this study—nor are memory output dependencies.

6.3.1 Total Ordering of Loads and Stores

The simplest method of issuing loads and stores is to keep them totally ordered with respect to each other-though not necessarily with respect to other instructions-as shown in Figure ***63.*** In Figure 63, the STORE v instruction is released in cycle 4 by the execution of all previous instructions. Because loads are held until all previous stores are issued, there is no need to detect dependencies between loads and stores. The disadvantage of this organization is that loads are held for stores, and stores in turn are held for all previous instructions. This serializes computations that involve memory accesses whenever a **store** appears in the instruction stream, and also causes decoder stalls because the load/store reservation station is full more often.

6.3.2 Load Bypassing of Stores

To overcome the performance limitations of total ordering of loads and stores, loads can be allowed to bypass stores as illustrated in Figure 64. In this example, all loads but one are

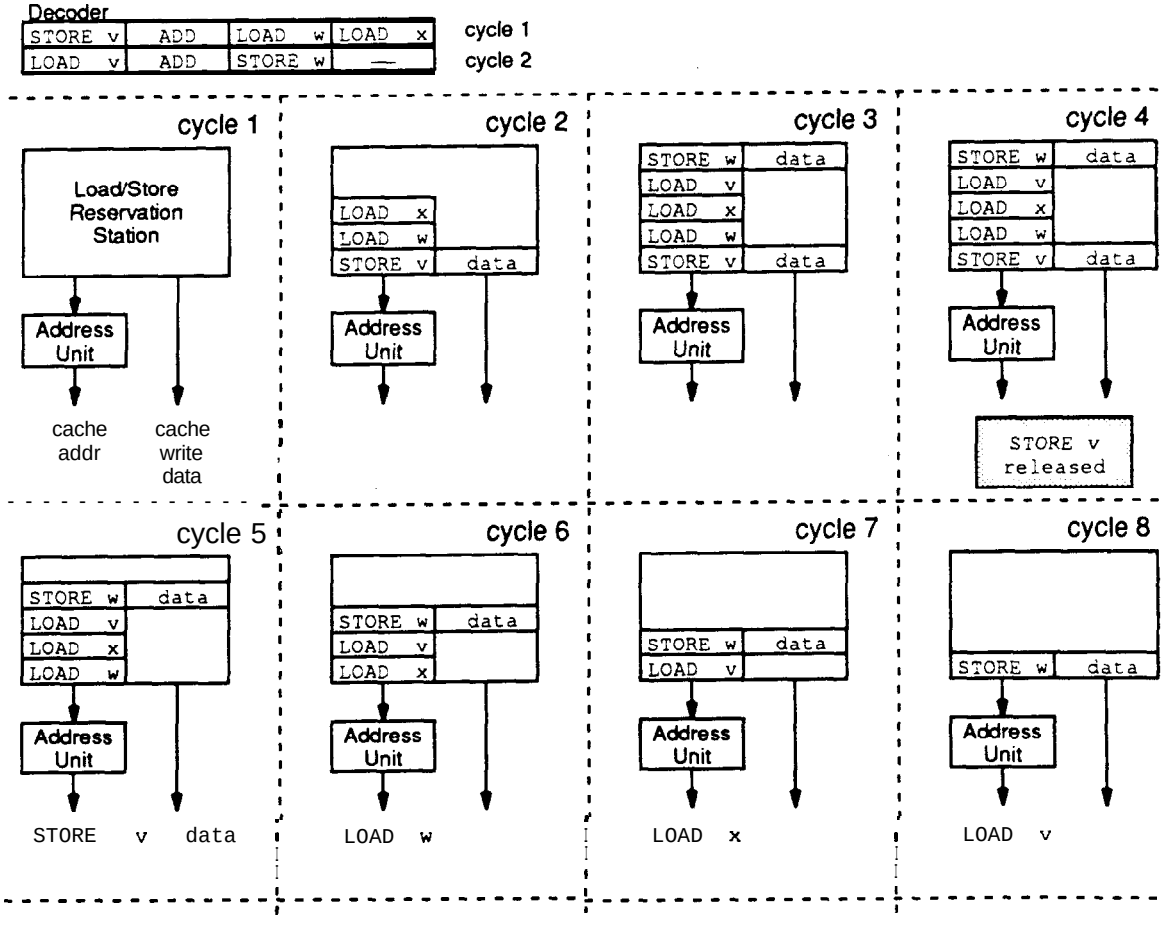

Figure 63. Issuing Loads and Stores with Total Ordering

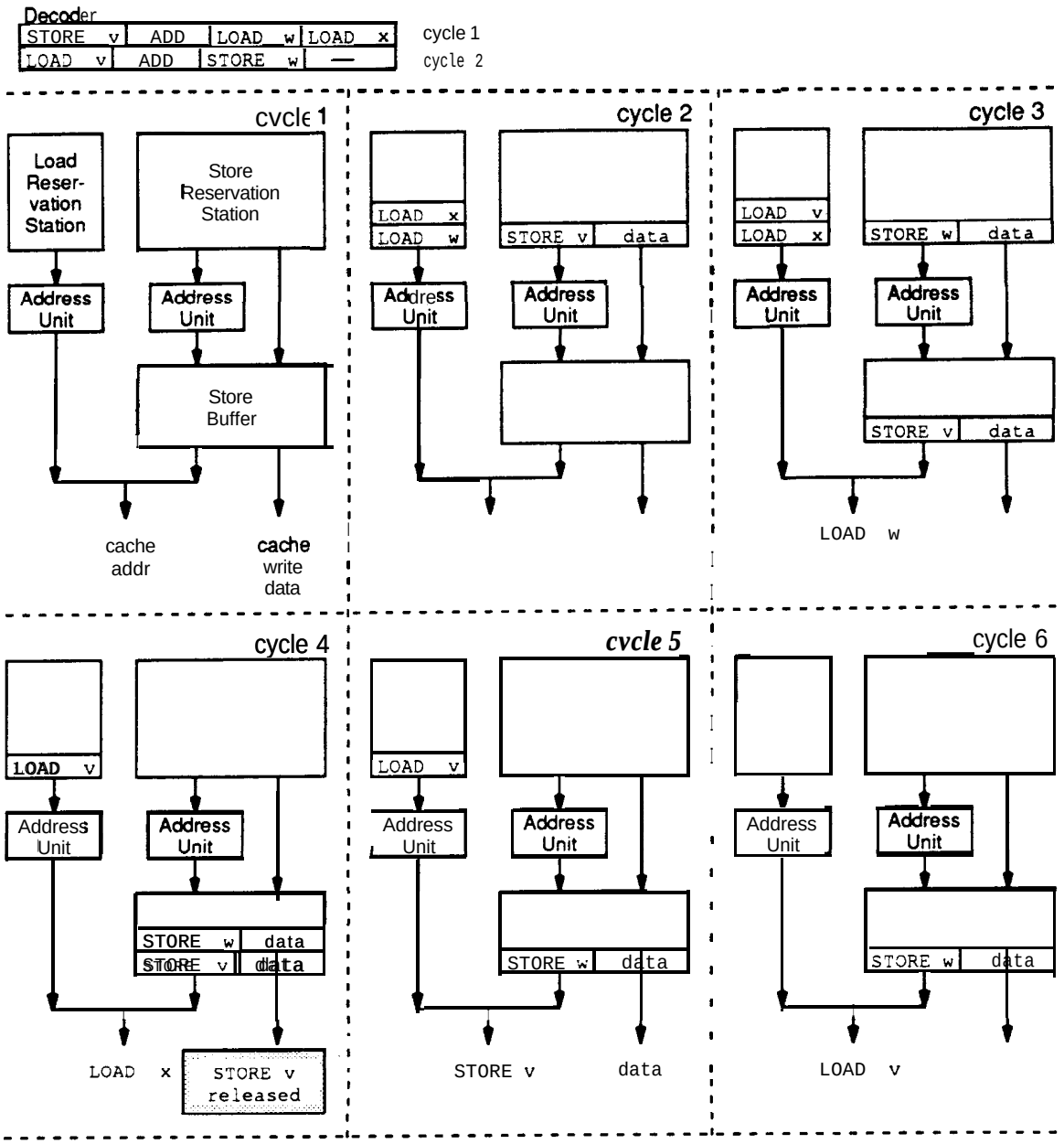

Figure 64. Load Bypassing of Stores

issued before the STORE v instruction is released for issue, even though the loads follow the STORE v in the instruction sequence. To support load bypassing, the hardware organization shown in Figure 64 includes separate reservation stations and address units for loads and stores and a store buffer (Section 6.3.5 below describes an organization that eliminates the reservation stations, using a central window and a single address unit).

The store buffer is required because a store instruction can be in one of two different stages of issue: before address computation and translation have been performed, and after addressing computation and translation but before the store is committed to the data cache. Store addresses must be computed before loads are issued so that dependencies can be checked, but stores must be held after address computation until previous instructions complete. Figure 64 shows that the dependent `LOAD` v instruction is not issued until the `STORE` v instruction is issued, even though preceding loads are allowed to bypass this store. If a store address cannot be determined (for example, because the address base register is not valid), all subsequent loads are held until the address is valid, because there might be a memory dependency.

6.3.3 Load Bypassing with Forwarding

When load bypassing is implemented, it is not necessary to hold a dependent load until the previous store is issued. As illustrated in Figure 65, the load can be satisfied directly from the store buffer if the address is valid and the data is available in the store buffer. This optimization avoids the additional latency caused by holding the load until the store is issued, and can help in situations where the compiler is forced by potential dependencies to keep operand values in memory (for example, in a program containing pointer de-references or address operators). In such situations, it is possible to have frequent loads of values recently stored.

6.3.4 Simulation Results

Figure 66 shows the results of simulating the benchmark programs using the various load/store mechanisms described in this section, for both two-instruction and four-instruction decoders. The interpretation of the chart labels is as follows:

- total order — total ordering of loads and stores.
- load byp — load bypassing of stores.
- load fwd — load bypassing of stores with forwarding.

Load bypassing yields an appreciable performance improvement: 11% with a two-instruction decoder, and 19% with a four-instruction decoder. Load forwarding yields a much smaller improvement over load bypassing: 1% with a two-instruction decoder and 4% with a four-instruction decoder. The obvious conclusion is that load/store mechanism should support load bypassing. Load forwarding is a less-important optimization that may or may not be justified, depending on implementation costs.

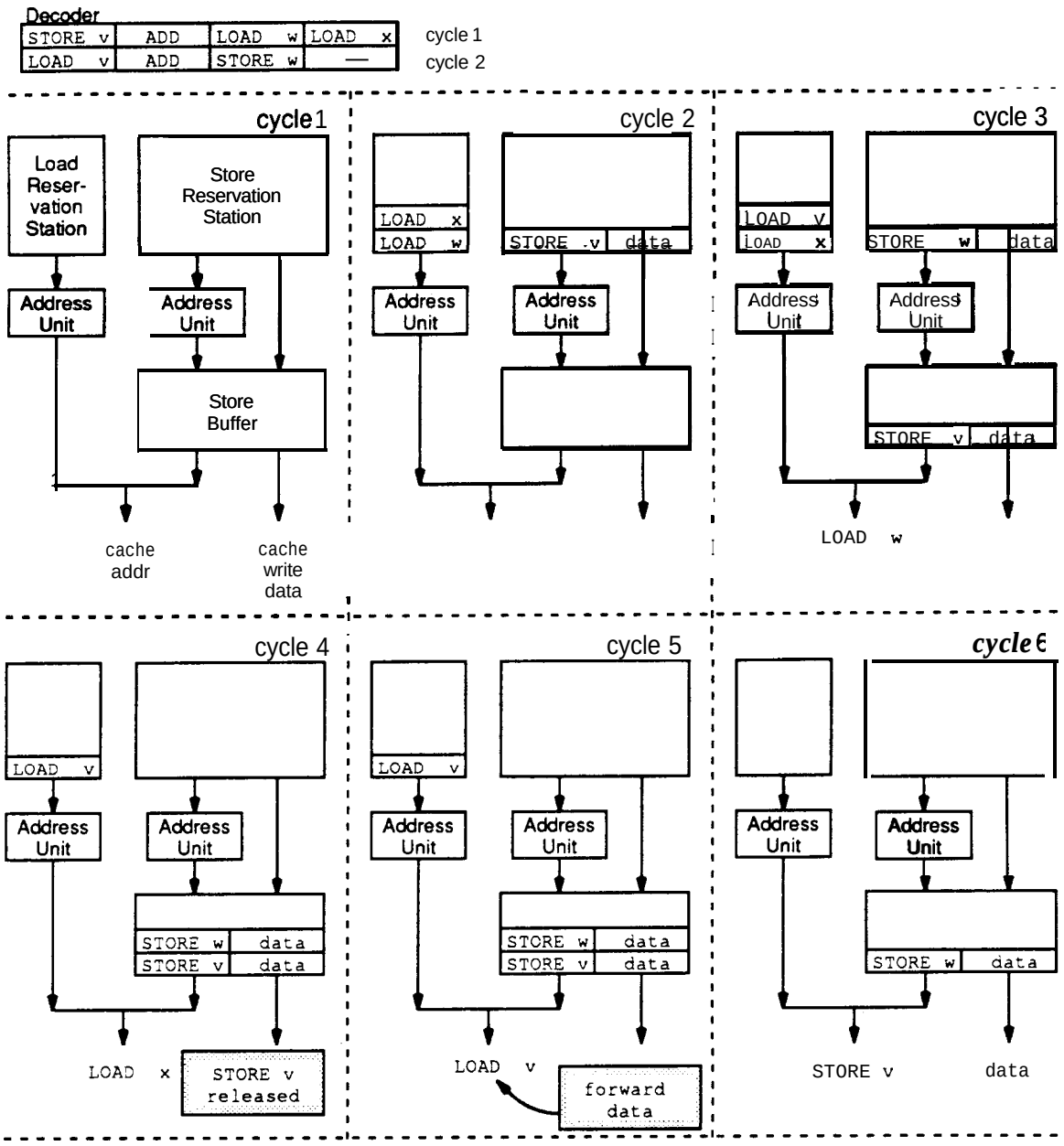

Figure 65. *Load* ***Bypassing*** *of* ***Stores with Forwarding***

6.3.5 Implementing Loads and Stores with a Central Instruction Window

The simulation results for load bypassing and load forwarding in Section 6.3.4 used the hardware organization shown in Figure 64. This organization is based on reservation stations to provide the most flexibility in the load and store mechanisms, bu this is an inconvenient implementation for two reasons. First, it assumes two separate address units each consisting of a 32-bit adder and (possibly) address-translation hardware. Two address units

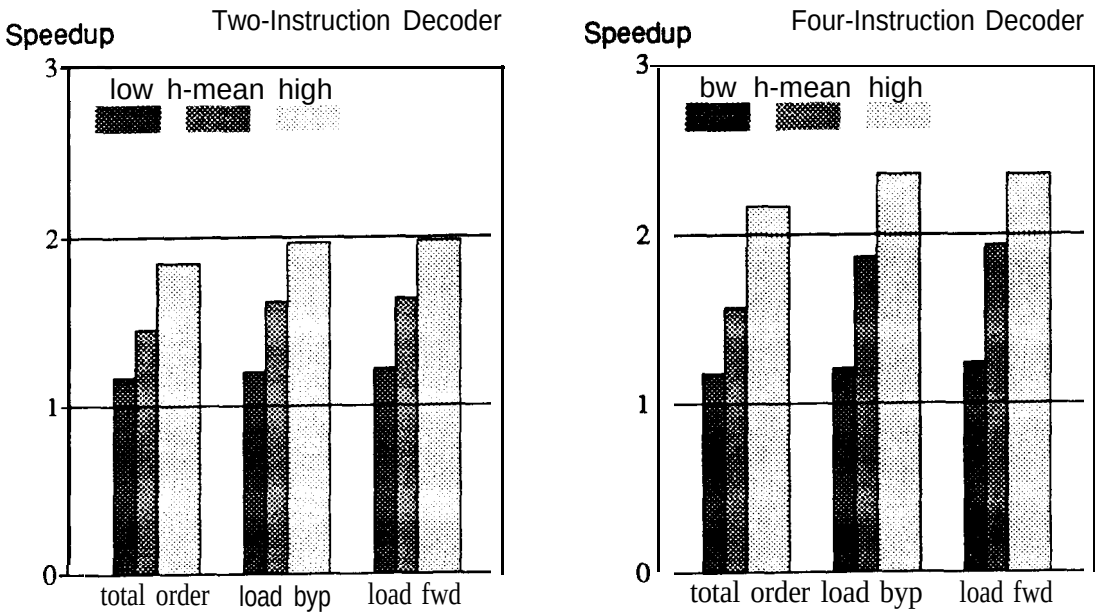

Figure 66. Performance for Various Load/Store Techniques

are largely unnecessary, because the issue rate for loads and stores is constrained by the data-cache interface to one access per cycle. Second, this organization makes it difficult to determine the original order of loads and stores so that dependency checking can be performed.

For example, when a load is issued, it may have to be checked against a store in the store reservation station, in the store address unit, or in the store buffer. Stores in the store reservation station have not had address computation and translation performed. If a load sequentially follows a store that is still in the store reservation station, the load must be held at least until the store is placed into the store buffer. However, there is no information in the reservation stations or store buffer, as defined, to allow the hardware to determine the set of stores on which a given load may depend. Without a mechanism to determine the correct set of stores, deadlock can occur: a store may hold because of a register dependency on a previous load, while the load holds because it is considered (incorrectly) to depend on the store.

An alternative organization to that shown in Figure 64 is shown in Figure 67. This organization is based on a central window and has a single address unit. Loads and stores are issued to the address unit from the central window. A store buffer operates as before to hold stores until they are committed to the data cache.

When a load or store instruction is decoded, its address-register value (or a tag) and address offset are placed into the central window. The data for a store (or a tag) is placed directly into

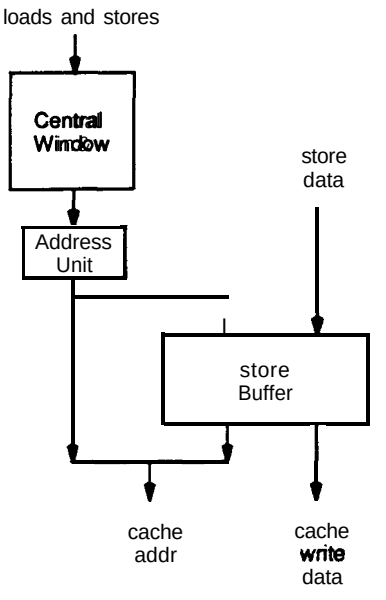

Figure 67. Reorganizing the Load/Store Unit with a Central Instruction Window

the store buffer. From the central window, loads and stores are issued in original program order to the address unit (this can be accomplished using sequencing tags, as suggested in Section 6.2.3). A load or store cannot be issued from the window until its address-register value is valid, as usual. At the output of the address unit, a store address is placed into the store buffer, and a load address is sent directly to the data cache (the load bypasses stores in the store buffer). Loads are checked for dependencies against stores in the store buffer. If a dependency exists, load data can be forwarded from the store buffer.

Figure 68 repeats the simulation results of Section 6.3.4 for this organization (the results for totally-ordered loads and stores are shown for continuity, but there is no difference in this case). For these results, an eight-entry central window and an eight-entry store buffer were used. In most cases, there is only a slight performance penalty caused by issuing loads to the address unit in order with stores-l % with a two-instruction decoder and 3% with a four-instruction decoder. For the results in Section 6.3.4, a load could be issued to the data cache while a store was simultaneously placed into the store buffer. Since this latter technique requires two address units, and since the central-window approach does not reduce performance very much, the implementation using the central window and a single address is a good cost/performance tradeoff.

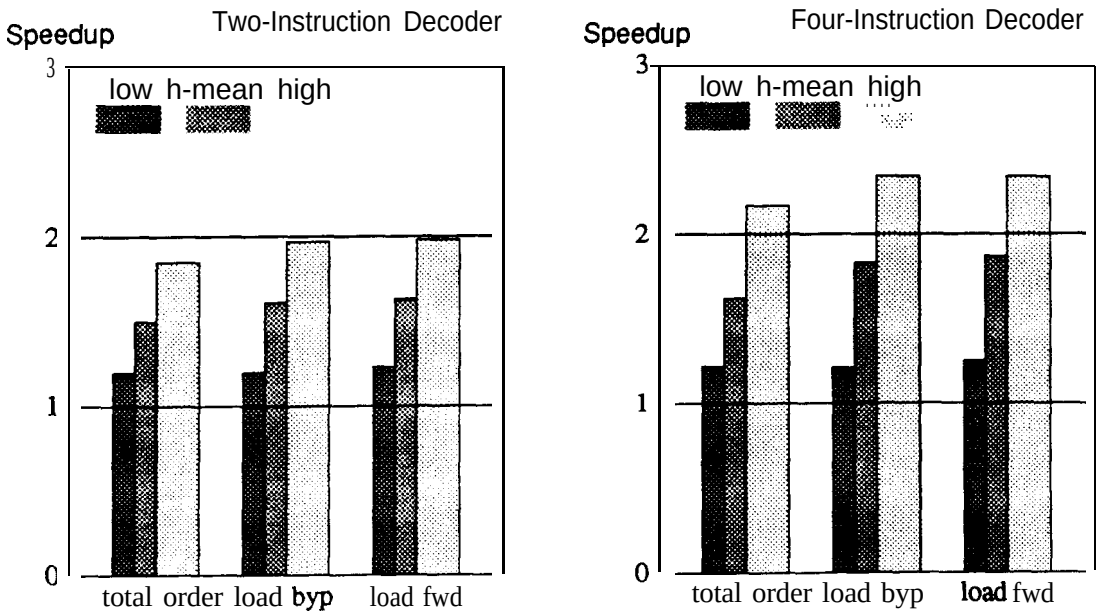

Figure 68. *Performance for Various Load/Store* ***Techniques using a Central Window***

6.3.6 Effects of Store Buffer Size

Figure 69 shows the effects of store-buffer size on performance, for a processor having an eight-entry central window and implementing load bypassing with forwarding. With a two-instruction decoder, both a four-entry and a two-entry store buffer have nearly identical performance to an eight-entry buffer. With a four-instruction decoder, a four-entry buffer causes a 1% performance loss over an eight-entry buffer, and a two-entry buffer causes a 4% loss. The four-entry buffer is a good choice for the four-instruction decoder, because it has good performance and the small number of entries facilitate dependency checking.

6.3.7 Memory Dependency Checking

The organization of Figure 67 greatly simplifies memory dependency checking, in comparison to the organization in Figure 64, because loads and stores are issued to the address unit in order. When a load is issued, its address is checked for dependencies against all valid addresses in the ***store*** buffer. The store buffer contains only stores that sequentially preceded the load, and the store buffer is the only location for these stores until they are issued, after which dependency checking no longer matters.

If a load address matches an address in the store buffer and load forwarding is implemented, the load data are supplied, if available, directly from the store buffer. The dependency logic

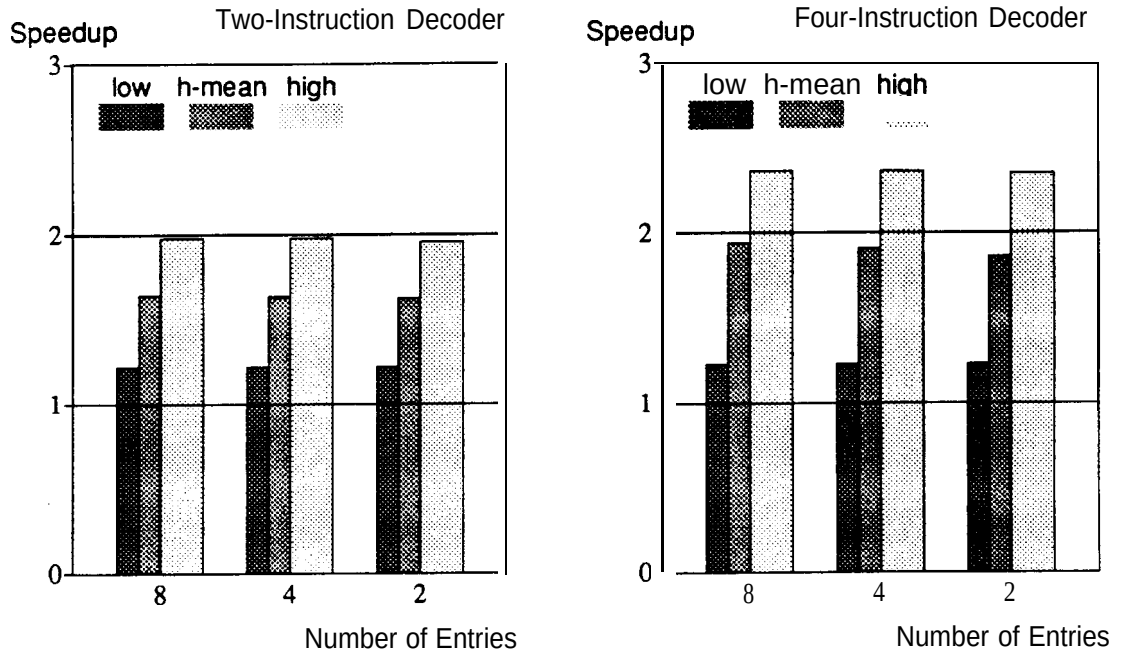

Figure 69. *Effect of Store-Buffer Sizes*

must identify the most recent matching store-buffer entry, because the address of more than one store-buffer entry may match the load address. The most recent entry contains the required data (or a tag for these data). If forwarding is not implemented, the load is held if its address matches the address of any store-buffer entry, without regard to the number of matching entries, until all matching store-buffer entries are committed to the data cache.

To reduce the logic chains involved in dependency checking and forwarding, these operations can be performed as a load is issued to the data cache, with the load being squashed or canceled as required if a dependency is detected. Squashing and canceling may require support in the data cache, and these operations are possible only if loads are free of side-effects in the system.

If load forwarding is not implemented, the comparators used to detect memory dependencies can be reduced by checking a subset of the full address. Figure 70 shows how performance is reduced as dependency checking is limited to 8, 4, 2, and 1 address bits. Although limiting address checking in this manner sometimes causes false dependencies (for these results, low-order address bits were compared under the assumption that these bits have the most variation and are least likely to compare falsely), the effect on performance is very small until less than 8 bits are used. These results indicate that 8-bit comparators cause

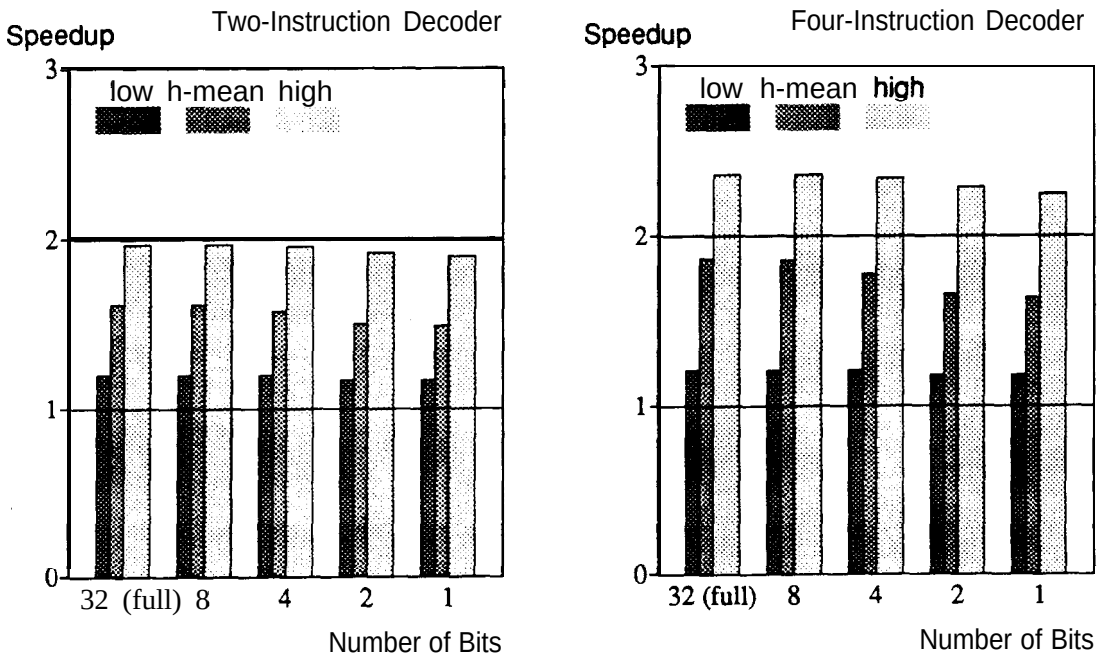

Figure 70. *Effect of* ***Limiting Address Bits*** *for Memory* ***Dependency Checking***

negligible reduction in performance. Note, however, that some performance is lost because load forwarding is not possible with this technique.

If both load forwarding and address translation are implemented, the size of the dependency checking comparators can be reduced using information from the address-translation operation. If the address-translation system does not allow a process to creates aliases for memory locations via address translation, the identifier of the translation look-aside buffer (TLB) entry used to translate an address-concatenated with the page offset of the address-provides a shortened form of the address that can be used in dependency checking. Under these conditions, comparisons involving these shortened addresses are exact. As an example of the savings, a 32-entry TLB and 4-Kbyte virtual page size allow the dependency comparators to be reduced to 17 bits.

6.4 Observations and Conclusions

Out-of-order issue can be accomplished either by reservation stations or by a central instruction window. Reservation stations hold instructions for issue near to a particular functional unit, distributing and partitioning the tasks involved in issuing instructions. In contrast, the central window maintains all pending instructions in one central location, examining all instructions in every cycle to select candidates for issue. The central window is more storage-

efficient, and consolidates all of the scheduling logic into a single unit. However, the scheduling logic is more complex than the scheduling logic of any given reservation station, primarily because there are more instructions in the central window than in any single reservation station and because functional-unit conflicts must be resolved by the central window. Still, overall considerations weigh in favor of the central window. An alternative to two published central-window proposals-the dispatch stack and register update unit-relies on the reorder buffer to maintain instruction-sequencing information. This is one of several examples of synergy between hardware components in the super-scalar processor.

Loads and stores present several impediments to high instruction throughput, because of issue constraints, addressing operations, and hardware interfaces associated with memory accesses. Requirements of correctness and restartability reduce the opportunities to gain concurrency by scheduling loads and stores out-of-order. Furthermore, the hardware required to achieve additional concurrency is expensive in relation to the performance benefits provided by this hardware. Compiler scheduling of loads and stores is not as effective as for other instructions (for the general-purpose benchmarks used in this study), because the compiler has little knowledge of memory dependencies that can arise during execution.

Of all the alternatives examined in this chapter, an organization implementing load bypassing (without forwarding) with a central window and a store buffer provides the best tradeoff of hardware complexity and performance. This organization requires a single address unit that is used by loads and stores in their original program order. Memory dependency checking can be performed with eight low-order bits of the memory address with no appreciable loss in performance, and dependency checking is not concerned with the order of stores in the store buffer as it is when load forwarding is implemented. The store buffer requires only four entries. The performance with this organization is not the best that can be achieved, but is still good, because it allows loads to be issued out-of-order with respect to stores.

Chapter 7
Conclusion

This research has used an experimental approach to architecture design: processor features are evaluated in light of the performance they provide. Trace-driven simulation has permitted the evaluation of a large number of super-scalar design alternatives for a large sample of general-purpose benchmark applications. This approach has identified major hardware components that are required for best performance and has uncovered techniques that simplify hardware with little reduction in performance. By defining the limits of super-scalar hardware, by illustrating the complexity of super-scalar hardware, and by suggesting possible software support, this research has established a basis for future research into hardware and software tradeoffs for super-scalar processor design.

This thesis shows that a general-purpose, super-scalar processor needs four major hardware features for best performance: out-of-order execution, register renaming, branch prediction, and a four-instruction decoder. The complexity of these features-although not unmanageable-may argue against the goal of achieving highest possible performance. However, hardware complexity must be considered in view of hardware simplifications which have been proposed throughout this study. This chapter summarizes these simplifications and emphasizes that-even if hardware is simplified in many ways-performance is not reduced as much as when one of the four major features list above is removed.

This study has also pointed out several avenues for further research, primarily in the direction of software support. The general-purpose benchmark applications used in this study do not lend themselves to software scheduling as readily as many scientific applications, but there are still many ways in which software can help simplify hardware and improve performance.

7.1 Major Hardware Features

Table 6 summarizes the performance advantages of out-of-order execution, register renaming, branch prediction, and a four-instruction decoder. Each entry in Table 6 is the relative performance increase due to adding the given feature, in a processor that has all other listed features. This simple summary indicates that each of these features is important in its own right. However, these features are interdependent in ways not illustrated by Table 6.

Branch prediction and a four-instruction decoder overcome the two major impediments to supplying an adequate instruction bandwidth for out-of-order execution: fetch stalls due to

Table 6. Performance Advantage of Major Processor Features

Out-of-order execution	Register renaming	Branch prediction	Four-instruction decoder
52%	36%	30%	18%

branch resolution and decoder inefficiency caused by instruction misalignment. Techniques to overcome the branch delay in scalar processors, such as compiler scheduling of branch delays, are not effective in a super-scalar processor for general-purpose applications because of the frequency of branches and the magnitude of the branch-delay penalty. With a speedup of two, the branch delay represents about eight instructions. With branch prediction, a four-instruction decoder achieves higher performance than a two-instruction decoder, because the four-instruction decoder supplies adequate instruction bandwidth even in the face of misaligned instruction runs.

Out-of-order execution and register renaming not only provide performance in the expected ways, but also provide performance by supporting branch prediction. The instruction window for out-of-order execution provides a buffer in which the instruction fetcher can store instructions following a predicted branch, and register renaming provides a mechanism for recovering from mispredic ted branches.

Table 6 implies another way in which out-of-order execution, register renaming, branch prediction, and a four-instruction decoder depend on each other. These features together provide more performance than is provided by each feature taken separately, because each relative improvement in Table 6 assumes that the other features are already implemented. Thus, it is difficult to justify implementing anything but the complete set of features, except that this observation presupposes that cycle time is not affected and that the performance goal justifies the design complexity.

The complexity of the super-scalar processor is its most troubling aspect. None of the major features by itself is particularly difficult to implement, but interdependencies between these features can create complex hardware and long logic delays. Complexity is significantly increased by the goal of decoding, issuing, and executing more than one instruction per cycle. Often, this complexity manifests itself in the arbitration logic that allocates multiple, shared processor resources among multiple contenders. The super-scalar processor has a number of different arbiters, each with its own complexity and timing constraints. The

design of these arbiters has been widely ignored in the published literature and in this study because arbiter design is very dependent on the hardware technology and implementation.

7.2 Hardware Simplifications

Though a super-scalar processor is complex, this study has suggested that many hardware simplifications are possible. This section summarizes these simplifications and their effects on performance. Although it is possible that, when a super-scalar processor is simplified in a number of ways, the cumulative effect of these simplifications reduces performance more than removing one of the major processor features listed in Section 7.1, this section shows that this is not the case.

Table 7 lists several progressively-simplified processor designs, showing the cumulative performance degradation. The cumulative degradation of each design is the percentage reduction in performance (based on the harmonic mean) relative to the processor configuration described in Section 3.3. Figure 7 1 and Figure 72 present the speedups of the configurations in Table 7. Although one conclusion presented in Section 7.1 was that a four-instruction decoder provides better performance than a two-instruction decoder, data for the two-

Table 7. Cumulative Effects of Hardware Simplifications

	Hardware Simplification (cumulative)	Degradation (cumulative)	
		2-inst	4-inst
a	execution model described in Section 3.3, with branch prediction in instruction cache	n/a[1]	n/a[2]
b	single-port array in instruction cache for branch-prediction information (Section 4.3.4)	-3%	-4%
c	limiting decoder to four register-file read ports (Section 4.4) and decoding a single branch instruction per cycle (Section 4.5.3)	-3%	-6%
d	waiting until mispredicted branch reaches the head of the reorder buffer before restarting (Section 5.2.1)	-7%	-11%
e	central instruction window (not compressed--Section 6.2.3), limited operand buses (Section 6.2.4), single address unit (Section 6.3.5)	-8%	-15%
f	no load forwarding from store buffer (Sections 6.3.2 and 6.3.3), 8-bit dependency checking (Section 6.3.7)	-9%	-17%
g	no load bypassing of stores (Section 6.3.1)	-16%	-28%

[1]speedup = 1.64 [2]speedup = 1.94

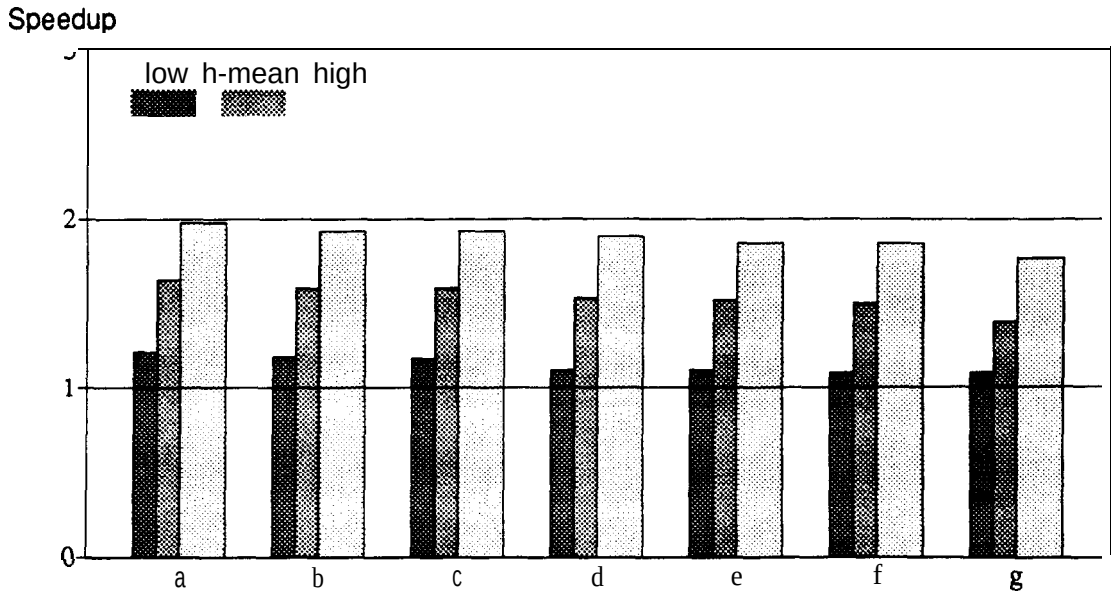

Figure 71. Cumulative Simplifications with Two-Instruction Decoder

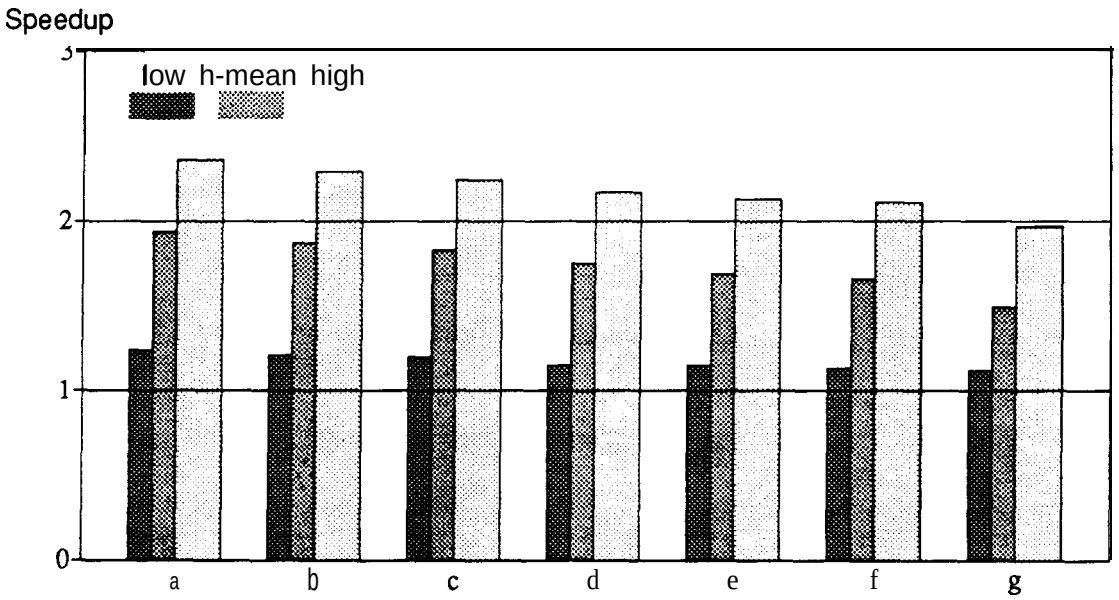

Figure 72. Cumulative Simplifications with Four-Instruction Decoder

instruction decoder is presented here for two reasons. First, it shows the effects of hardware simplifications in a processor that is more severely instruction-fetch limited. Second, it illustrates that a four-instruction decoder still achieves better performance than a two-instruction decoder even with simplified hardware.

Many of the hardware simplifications do not reduce performance very much because the instruction window decouples instruction-fetch and instruction-execution limitations. The average rates of fetching and execution are more important to overall performance than short-term limitations. Other simplifications are the direct result of the hardware features described in Section 7.1. For example, the instruction-ordering information retained by the reorder buffer allows this information to be discarded as instructions are allocated entries in the instructions window, avoiding the need to compress the window and simplifying its implementation. As a further example, the dynamic scheduling of the instruction window allows good performance with a limited number of operand buses between the window and the functional units.

Eliminating load bypassing is the only simplification listed in Table 7 that causes lower performance than the removal of out-of-order execution, register renaming, branch prediction, or a four-instruction decoder. Apparently, the conclusion of Section 6.3—that load bypassing is the most important feature of the load/store mechanism-is still valid in a simplified processor.

7.3 Future Directions

This study has suggested several areas where software support can improve processor performance or simplify the hardware. The suggested support raises several important questions.

The instruction format proposed in Section 4.4.2 relies on software to schedule the use of register-file read ports. Software can either use common source registers among several instructions or use dependent instructions to reduce the demands on register-file read ports. Whether either approach is feasible is an open question. Furthermore, using dependent instructions to reduce register-file demands conflicts with the overall software-scheduling goal of helping the processor to fetch and decode independent instructions. It is difficult to determine without much further study the degree to which these goals conflict.

Another proposal in this study is that software can improve the efficiency of the instruction fetcher by loop unrolling, aligning instruction runs, and scheduling instructions after a branch to pad the decoder. Hardware-based aligning and merging indicates that these can be important optimizations, but further research is required to determine whether software can perform aligning and merging with an overall benefit, particularly with respect to the effect of the increased code density on the hit ratio of the instruction cache.

Finally, though this study has established a starting point for future research, many of the conclusions have been reached with the assumption of no software support. Further study may show that more hardware should be provided to exploit the instruction independence that software is able to provide, or that less hardware is required because software is able to provide benefits that were not anticipated in this study.

References

[Acosta et al. 1986]

R.D. Acosta, J. Kjelstrup, and H.C. Tomg, "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors." ***IEEE Transactions*** on ***Computers,*** Vol. C-35, (September 1986), pp. 815-828.

[Agerwala and Cocke 19871

T. Agerwala and J. Cocke, "High Performance Reduced Instruction Set Processors." Technical Report RC12434 (#55845), IBM Thomas J. Watson Research Center, Yorktown, NY, (January 1987).

[Backus 1978]

J. Backus, "Can Programming Be Liberated From the Von Neumann Style? A Functional Style and Its Algebra of Programs." ***Communications of the ACM,*** Vol. 21, No. 8, (August 1978), pp. 613641.

[Chaitin et al. 1981]

G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein, "Register Allocation via Coloring." ***Computer Languages,*** Vol. 6, No. 1 (1981), pp.47–57.

[Charlesworth 19811

A.E. Charlesworth, "An Approach to Scientific Array Processing: The Architectural Design of the AP–120B/FPS 164 Family." ***Computer,*** Vol. 14, (September 1981), pp. 18-27.

[Colwell et al. 1987]

R.P. Colwell, R.P. Nix, J.J. O'Donnel, D.B. Papworth, and P.K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," ***Proceedings of the Second International Conference on Architectural Supportfor Programming Languages and Operating Systems,*** (October 1987), pp. 180-192.

[Ditzel and McLellan 1987]

D.R. Ditzel and H.R. McLellan, "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero." ***Proceedings of the 14 th Annual Symposium on Computer Architecture,*** (June 1987), pp. 2-9.

[Dwyer and Torng 19871

H.C. Tomg, "A Fast Instruction Dispatch Unit for Multiple and Out-of-Sequence Issuances." School of Electrical Engineering Technical Report EE–CEG–87–15, (November 1987), Cornell University, Ithaca, NY.

[Fisher 1981]

J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction." ***IEEE Transactions on Computers,*** Vol. C-30, (July 1981), pp. 478-490.

[Fisher 1983]

J.A. Fisher, "Very Long Instruction Word Architectures and the ELI-512." ***Proceedings*** *of* ***the 10th Annual Symposium on Computer*** *Architecures*, (June 1983), pp. 140-150.

[Foster and Riseman 1972]

C.C. Foster and E.M. Riseman, "Percolation of Code to Enhance Parallel Dispatching and Execution." ***IEEE Transactions on Computers,*** Vol. C-21, (December 1972), pp. 1411-1415.

[Gross and Hennessy 1982]

T.R. Gross and J.L. Hennessy, "Optimizing Delayed Branches." ***Proceedings*** *of* ***IEEE*** *Micro-15*, (October 1982), pp. 114-120.

[Hennessy and Gross 1983]

J. Hennessy and T. Gross, "Postpass Code Optimization of Pipeline Constraints." ***ACM Transactions on Programming Languages and Systems,*** Vol. 5, No. 3, (July 1983), 422–448.

[Hennessy 19861

J.L. Hennessy, "RISC-Based Processors: Concepts and Prospects." ***New Frontiers in Computer Architecture Conference Proceedings,*** (March 1986), pp. 95-103.

[Hwu and Chang 1988]

W.W. Hwu and P.P. Chang, "Exploiting Parallel Microprocessor Microarchitectures with a Compiler Code Generator." ***Proceedings*** *of* ***the 15th Annual Symposium on Computer Architecture,*** (June 1988), pp. 45-53.

[Hwu and Patt 1986]

W. Hwu and Y.N. Patt, "HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality." ***Proceedings of the 13 th Annumal Symposium on Computer Architecture,*** (June 1986), pp. 297-307.

[Hwu and Patt 1987]

W.W. Hwu and Y.N. Patt, "Checkpoint Repair for Out-of-Order Execution Machines." ***Proceedings of the 14th Annual Symposium on Computer Architecture,*** (June 1987), pp. 18-26.

[Jouppi and Wall 1988]

N.P. Jouppi and D.W. Wall, "Available Instruction-Level Parallelsim for Superscalar and Superpipelined Machine." Technical Note TN-2, Digital Equipment Corporation Western Research Laboratory , Palo Alto, CA, (September 1988). Also published in ***Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems,*** (April 1989), pp. 272-282.

[Keller 1975]

R.M. Keller, "Look-Ahead Processors." ***Computing Surveys,*** Vol. 7, No. 4, (December 1975), pp. 177-195.

[Lam 1988]

M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines." ***Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, (June 1988),*** pp. 3 18-328.

[Lee and Smith 1984]

J.K.F. Lee and A.J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design." ***IEEE Computer,*** Vol. 17, (January 1984), pp. 6-22.

[McFarling and Hennessy 1986]

S. McFarling and J. Hennessy, "Reducing the Cost of Branches." ***Proceedings of the 13th Annual Symposium on Computer Architecture,*** (June 1986), pp. 396-404.

[MIPS 86]

MIPS Computer Systems, Inc., ***MIPS Language Programmer's Guide,*** (1986).

[Mueller et al. 1984]

R.A. Mueller, M.R. Duda, and S.M. O'Haire, "A Survey of Resource Allocation Methods in Optimizing Microcode Compilers." ***Proceedings of the 17th Annual Workshop on Microprogramming,*** (October 1984), pp. 285-295.

[Nicolau and Fisher 19841

A. Nicolau and J.A. Fisher, "Measuring the Parallelism Available for Very Long Instruction Word Architectures." ***IEEE Transactions on Computers,*** Vol. C-33, (November 1984), pp. 968-976.

[Patt et al. 1985a]

Y.N. Patt, W. Hwu, and M. Shebanow, "HPS, A New Microarchitecture: Rationale and Introduction." ***Proceedings of the 18th Annual Workshop on Microprogramming,*** (December 1985), pp. 103-108.

[Patt et al. 1985b]

Y.N. Patt, S.W. Melvin, W. Hwu, and M.C. Shebanow, "Critical Issues Regarding HPS, A High Performance Microarchitecture." ***Proceedings of the 18th Annual Workshop on Microprogramming,*** (December 1985), pp. 109-l 16.

[Pleszkun et al. 1987]

A. Pleszkun, J. Goodman, W.C. Hsu, R. Joersz, G. Bier, P. Woest, and P. Schecter, "WISQ: A Restartable Architecture Using Queues." ***Proceedings of the 14th Annual Symposium on Computer Architecture,*** (June 1987), pp. 290-299.

[Pleszkun and Sohi 1988]

A.R. Pleszkun and G.S. Sohi, "The Performance Potential of Multiple Functionand Unit Processors." ***Proceedings of the 15th Annual Symposium on Computer Architecture,*** (June 1988), pp. 3744.

[Przybylski et al. 1988]

S. Przybylski, M. Horowitz, and J. Hennessy, "Performance Tradeoffs in Cache Design." ***Proceedings of the 15th Annual Symposium on Computer Architecture,*** (June 1988), pp. 290-298.

[Rau and Glaeser 1981]

B.R. Rau and C.D. Glaeser, "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing." ***Proceedings of the 14th Annual Workshop on Microprogramming,*** (October 1981), pp. 183-198.

[Riseman and Foster 1972]

E.M. Riseman and C.C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps." ***IEEE Transactions on Computers,*** Vol. C-21, (December 1972), pp. 1405-1411.

[Sohi and Vajapeyam 1987]

G.S. Sohi and S. Vajapeyam, "Instruction Issue Logic for High-Performance Interruptable Pipelined Processors." ***Proceedings of the 14th Annual Symposium on Computer Architecture,*** (June 1987), pp. 27-34.

[Smith et al. 1988]

M.D. Smith, M. Johnson, and M.A. Horowitz, "Limits on Multiple Instruction Issue." ***Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems,*** (April 1989), pp. 290-302.

[Smith and Pleszkun 1985]

J.E. Smith and A.R. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors." ***Proceedings of the 12th Annual International Symposium on Computer Architecture,*** (June 1985), pp. 36-44.

[Thorton 1970]

J.E. Thorton, ***Design of a Computer—The Control Data*** 6600. Scott, Foresman and Co., Glenview IL, (1970).

[Tjaden and Flynn 1970]

G.S. Tjaden and M.J. Flynn, "Detection and Parallel Execution of Independent Instructions." ***IEEE Transactions on Computers,*** Vol. C-19, No. 10, (October 1970), pp. 889-895.

[Tjaden 1972]

G.S. Tjaden and M.J. Flynn, "Representation and Detection of Concurrency using Ordering Matrices." Ph.D. Dissertation, The Johns Hopkins University, Baltimore, MD, (1972).

[Tjaden and Flynn 1973]

G.S. Tjaden and M.J. Flynn, "Representation of Concurrency with Ordering Matrices." ***IEEE Transactions on Computers,*** Vol. C-22, No. 8, (August 1973), pp. 752-761.

[Tomasulo 1967]

R.M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units." ***IBM Journal,*** Vol. 11, (January 1967), pp. 25-33.

[Torng 1984]

H.C. Tomg, "An Instruction Issuing Mechanism for Performance Enhancement." School of Electrical Engineering Technical Report EE-CEG-84-1, (February 1984), Cornell University, Ithaca, NY.

[Uht 1986]

A.K. Uht, "An Efficient Hardware Algorithm to Extract Concurrency From General-purpose Code." ***Proceedings of the Nineteenth Annual Hawaii Internation Conference on System Sciences,*** (1986), pp. 41-50.

[Wedig 1982]

R.G. Wedig, "Detection of Concurrency in Directly Executed Language Instruction Streams." Ph.D. Dissertation, Stanford University, Stanford, CA (June 1982).

[Weiss and Smith 1984]

S. Weiss and J.E. Smith, "Instruction Issue Logic in Pipelined Supercomputers." ***IEEE Transactions on Computers,*** Vol. C-33 (November 1984), pp. 1013-1022.

[Weiss and Smith 1987]

S. Weiss and J.E. Smith, "A Study of Scalar Compilation Techniques for Pipelined Supercomputers." ***Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems,*** (October 1987), pp. 105-109.

[Wulf 1988]

W.A. Wulf, "The WM Computer Architecture." ***Architecture News,*** (January 1988), pp. 70-84.