THE MULTISCALAR ARCHITECTURE

by

MANOJ FRANKLIN

A thesis submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN — MADISON

1993

THE MULTISCALAR ARCHITECTURE

Manoj Franklin

Under the supervision of Associate Professor Gurindar S. Sohi

at the University of Wisconsin-Madison

ABSTRACT

The centerpiece of this thesis is a new processing paradigm for exploiting instruction level parallelism. This paradigm, called the *multiscalar* paradigm, splits the program into many smaller tasks, and exploits fine-grain parallelism by executing multiple, *possibly (control and/or data) dependent* tasks in parallel using multiple processing elements. Splitting the instruction stream at statically determined boundaries allows the compiler to pass substantial information about the tasks to the hardware. The processing paradigm can be viewed as extensions of the superscalar and multiprocessing paradigms, and shares a number of properties of the sequential processing model and the dataflow processing model.

The multiscalar paradigm is easily realizable, and we describe an implementation of the multiscalar paradigm, called the multiscalar processor. The central idea here is to connect multiple sequential processors, in a decoupled and decentralized manner, to achieve overall multiple issue. The multiscalar processor supports speculative execution, allows arbitrary dynamic code motion (facilitated by an efficient hardware memory disambiguation mechanism), exploits communication localities, and does all of these with hardware that is fairly straightforward to build. Other desirable aspects of the implementation include decentralization of the critical resources, absence of wide associative searches, and absence of wide interconnection/data paths.

The multiscalar processor design has novel features such as the *Address Resolution Buffer (ARB)*, which is a powerful hardware mechanism for enforcing memory dependencies. The ARB allows arbitrary dynamic motion of memory reference instructions by providing an efficient way of detecting memory hazards, and instructing the multiscalar hardware to restart portions of the dynamic instruction stream when hazards are violated.

We also present the results of a detailed simulation study showing the performance improvements offered by the multiscalar processor. These preliminary results are very encouraging. The multiscalar paradigm seems to be an ideal candidate to reach our goal of 10+ instructions per cycle, and has tremendous potential to be the paradigm of choice for a circa 2000 (and beyond) ILP processor.

Acknowledgements

First of all, I offer praise and thanks with a grateful heart to my *Lord JESUS CHRIST*, to whom this thesis is dedicated, for *HIS* infinite love and divine guidance all through my life. Everything that I am and will ever be will be because of *HIM*. It was *HE* who bestowed me with the ability to do this piece of research. *HE* led and inspired me during the past six years over which I learned many spiritual matters, without which all these achievements are meaningless, and a mere chasing after the wind. So, to *God* be the praise, glory, and honor, for ever and ever.

I am deeply indebted to my PhD advisor, Professor Gurindar S. Sohi, for the constant guidance he provided in the course of this research. Guri shared with me many of his ideas, and gave extremely insightful advice aimed at making me aim higher and higher in doing research. His ability to identify significant ideas, and suppress insignificant ones was invaluable. I feel honored and privileged to have worked with him for this thesis.

I am very grateful to my MS advisor, Professor Kewal K. Saluja, for his encouragement and advice throughout my graduate studies, and for the opportunity to do research with him in many areas. I am deeply indebted to Professor James R. Goodman for his help and encouragement in steering me into Computer Sciences. I would like to thank Professor Mark Hill for his insightful comments on this thesis. I am very grateful to Professor Wilson Pearson for providing all facilities at Clemson University, where I can continue my career.

Members of my family, each in his/her own way, have taught me and loved me far beyond what I could ever repay. I would like to acknowledge my father Dr. G. Aruldhas and mother Mrs. Myrtle Grace Aruldhas, for their unfailing concern, care and support throughout my life. They have always encouraged me to strive for insight and excellence, while staying in the spiritual path. A heartfelt thanks to my sister, Dr. A. Ann Adeline, for her continuous love and support over the years. I am grateful to my uncle, Dr. G. Sundharadas, for his help and encouragement, especially in the beginning stages of my graduate studies, a time when it was most needed.

I would like to acknowledge my friend Dr. B. Narendran, who kindled my interest in recreative mathematics and puzzles. Over the years, he has made many contributions to my papers on testing. I thank him for many helpful mathematical discussions and interesting puzzle solving sessions. On a different note, I thank my friend, Neena Dakua, for many stimulating philosophical discussions years back, which put me on a thinking path. I owe a special debt to her, D. Chidambara Krishnan, and my colleagues in Bangalore, for helping me to come to the USA for higher studies. I would like to acknowledge my friend Dr. Rajesh Mansharamani for taking care of all the logistics in connection with my thesis submission. Dionisios N. Pnevmatikatos took care of all my UNIX questions.

I would like to acknowledge my close friends in India, R. Rajendra Kumar and D. Sanil Kumar, who have demonstrated the reality of ‘‘friend in need’’, by the warmth and affection of their selfless love, which has accompanied me through the ups and downs of life. Rajendran’s letters from far away have always been a constant source of encouragement.

I would very much like to thank my brothers and sisters in Christ, for their prayers and care over the years in Madison. In particular, I would like to thank Chellapandi Subramanian, Adharyan T. Dayan, Dr. Suseela Rajkumar, Mathews Abraham, Shelin Mathews, Paul Lanser, Karin Lanser, Professor Wayne M. Becker, and the members of the Faith Community Bible Church and Indonesian Christian Fellowship. They have made my time in Madison meaningful and enjoyable. The daily prayer meetings with Chella will always stay in my mind.

Finally, the arrival of my better half Dr. Bini Lizbeth George into my life inspired me to come out of the dissertator cocoon. If not for her, I would still be analyzing every nook and corner of the multiscalar architecture. She has shared every nuance of the last two years with me, and I express my deep gratitude to her for her love and affection.

My research as a graduate student was mostly supported by IBM through an IBM Graduate Fellowship, and also by Professor Gurindar Sohi through his NSF grant (contract CCR-8706722).

Table of Contents

Chapter 1. Introduction

How to exploit parallelism from ordinary, non-numeric programs?

In spite of the tremendous increases in computing power over the years, nobody has ever felt a glut in computing power; the demand for computing power seems to increase with its supply. To satisfy this demand in the midst of fast approaching physical limits such as speed of light, scientists should find ever ingenious ways of increasing the computing power. The main technique computer architects use to achieve speedup is to do *parallel processing*. Today digital computing is at a point where clock speeds of less than 10ns are becoming the norm, and further improvements in the clock speed require tremendous engineering effort. An order of magnitude improvement in clock speed (to achieve clock cycles in the sub-nanosecond range) may be difficult, especially because of approaching physical limits. However, looking into the late 1990s and the early 2000s, there appears to be no end in sight to the growth in the number of transistors that can be put on a single chip. Technology projections even suggest the integration of 100 million transistors on a chip by the year 2000 [60], in place of the 2-4 million transistors that are integrated today. This has prompted computer architects to consider new ways of utilizing the additional resources for doing parallel processing.

The execution of a computer program involves *computation* operations and *communication* of values, constrained by *control* structures in the program. The basic idea behind parallel processing is to use multiple hardware resources to process multiple computations as well as multiple communication arcs in parallel so as to reduce the execution time, which is a function of the total number of computation operations and communication arcs, the cycle time, and the average number of computation nodes and communication arcs executed in a cycle. With the continued advance in technology, switching components have become progressively smaller and more efficient, with the effect that computation has become reasonably fast. Communication speed, on the other hand, seems to be more restricted by the effects of physical factors such as the speed of light, and has become the major bottleneck.

The parallelism present in programs can be classified into different types — regular versus irregular parallelism, coarse-grain versus fine-grain (instruction level) parallelism, etc. Regular parallelism, also known as *data parallelism*, refers to the parallelism in performing the same set of operations on different elements of a data set, and is very easy to exploit. Irregular parallelism refers to parallelism that is not regular, and is harder to exploit. Coarse-grain parallelism refers to the parallelism between large sets of operations such as subprograms, and is best exploited by a multiprocessor. Fine-grain parallelism, or instruction level parallelism refers to the parallelism between individual operations. Over the years, several processing paradigms, including some special purpose paradigms, have been proposed to exploit different types of parallelism. While we would like to stay away from special purpose architectures and be as general as possible, we would still like to carve out our niche, which is *instruction level parallelism* or *ILP* for short. This thesis explores the issues involved in ILP processing, and proposes and investigates a new architectural paradigm for ILP processing.

1.1. Why ILP Processors?

Why, in any case, must we investigate ever ingenious ways of exploiting ILP? The answer is that it is important to exploit parallelism, and many important applications exist (mostly non-numeric) in which ILP appears to be the only type of parallelism available. Exploitation of parallelism at the instruction level is very important to speed up such programs. Recent studies have confirmed that there exists a large amount of instruction-level parallelism in ordinary programs [18, 25, 36, 106, 159]. Even in other applications, no matter how much parallelism is exploited by coarse-grain parallel processors such as the multiprocessors, a substantial amount of parallelism will still remain to be exploited at the instruction level. Therefore, irrespective of the speedup given by a coarse-grain parallel processor, ILP processing can give additional speedups over that speedup. Thus, coarse-grain parallel processing and ILP processing complement each other, and we can expect future multiprocessors to be a collection of ILP processors.

Although ILP processing has been used in the highest performance uniprocessors for the past 30 years, it was traditionally confined to special-purpose paradigms for exploiting regular parallelism from numeric programs. In this thesis we place a strong emphasis on exploiting ILP from non-numeric programs, which mostly contain irregular parallelism. This is not to belittle the importance of numeric programs, which are the backbone of many theoretical and simulation studies in scientific applications. We feel that numeric programs have received substantial attention in the past, whereas non-numerical programs have received only a passing attention. The ILP machines developed so far — vector machines, the primal VLIW machines [55], etc. — have all specifically targeted numeric applications. Our motivation for targeting the difficult irregular parallelism in non-numeric programs is that if we are able to develop techniques to extract and exploit even small amounts of ILP from such programs (which are anathema for ILP), then the same collection of techniques can be migrated to the realm of numeric programs to get perhaps an order of improvement in performance!

As of today, no single ILP processing paradigm has shown a spectacular performance improvement for non-numeric programs. This research is an attempt not only to bridge that gap, but also to lay the foundation for future ILP processors.

1.2. Challenges

There are several issues to be tackled in developing a good ILP processing paradigm. First, there are different schools of thought on when the extraction of parallelism is to be done — at compile time or at run time. Each method has its own strengths and shortcomings. We believe that any processing model that relies entirely on compile-time scheduling or on run-time scheduling is very likely to fail because of inherent limitations of both. So the challenge is to use the right mix of compile time and run time extraction of parallelism. The alternatives differ widely, based on the extent to which this question is answered by the compiler or the hardware, and on the manner in which the compiler-extracted parallelism information is conveyed to the hardware.

Second, studies have found little ILP within a small sequential block of instructions, but significant amounts in large blocks [18, 25, 90, 159]. There are several inter-related factors that contribute to this. Because most programs are written in an imperative language for a sequential machine with a limited number of architectural registers for storing temporary values, instructions of close proximity are very likely to be data dependent, unless they are reodered by the compiler. This means that most of the parallelism can be found only amongst instructions that are further apart in the instruction stream. The obvious way to get to that parallelism is to establish a large window of instructions, and look for parallelism in this window.

The creation of the large window, whether done statically or dynamically, should be accurate. That is, the window should consist mostly of instructions that are *guaranteed* to execute, and not instructions that *might* be executed. Recent work has suggested that, given the basic block sizes and branch prediction accuracies for some common C programs, following a single thread of control while establishing a window may not be sufficient: the maximum ILP that can be extracted from such a window is limited to about 7 [90]. A more complex window, which contains instructions from multiple threads of control might be needed; analysis of the control dependence graph [38, 51] of a program can aid in the selection of the threads of control.

Another major challenge in designing the ILP hardware is to decentralize the critical resources in the system. These include the hardware for feeding multiple operations to the execution unit, the hardware for carrying out the inter-operation communication of the many operations in flight, a memory system that can handle multiple accesses simultaneously, and in a dynamically scheduled processor, the hardware for detecting the parallelism at run time.

1.3. Contributions of this Thesis

In this thesis we describe a new processing model called the *multiscalar paradigm*; it executes programs by means of the parallel execution of multiple tasks that are derived from a sequential instruction stream. This is achieved by considering a subgraph of the program's control flow graph to be a task, and executing many such tasks in parallel. The multiple tasks in execution can have both data dependencies and control dependencies between them. The execution model within each task can be a simple, sequential processor. As we will see later in this thesis, such an approach has the synergistic effect of combining the advantages of the sequential and the dataflow execution models, and the advantages of static and dynamic scheduling. Executing multiple subgraphs in parallel, although simple in concept, has powerful implications:

(1) The hardware mostly involves only the replication of ordinary processors. This allows the critical hardware resources to be decentralized by divide-and-conquer strategy, as will be seen in chapters 3-6. A decentralized hardware realization facilitates clock speeds comparable to contemporary implementations of single-issue processors. Furthermore, it allows expandability.

(2) The compiler can, as far as possible, attempt to divide the instruction stream into tasks at those points which facilitate the execution of control-independent code in parallel, thereby allowing multiple flows of control. Even if the compiler may not know the exact path that will be taken through a task at run time, it may be fairly sure of the next task that will be executed. Thus, the overall large window can be made very accurate.

(3) It helps to overlap the execution of blocks of code that are not guaranteed to be (data) independent. The compiler can, of course, attempt to pack dependent instructions into a task, and as far as possible create tasks that are independent so as to improve the processor performance. However, the processing paradigm does not require the tasks to be independent, and this is a significant advantage.

(4) Because the multiscalar paradigm considers a block of instructions as a single unit (task), the compiler can convey to the hardware more information such as inter-task register dependencies and control flow information. Thus, the hardware need not reconstruct some of the information that was already available at compile time.

(5) It helps to process and execute multiple branches in the same cycle, which is needed to exploit significant levels of ILP.

(6) It helps to exploit the localities of communication present in a program.

These statements may appear a bit ‘‘rough-and-ready’’, and may not make much sense before studying the new paradigm. It is precisely this paradigm and its implementation that we study in the ensuing chapters of this thesis.

1.4. Organization of the Thesis

We have outlined the important issues pertaining to the design of a high-performance ILP processor, and have now sketched in enough common ground for our study of ILP processing and the multiscalar paradigm to begin. The rest of this thesis is organized as follows. Chapter 2 presents a critique on ILP processing. It describes the fundamental concepts related to ILP processing, and explores how these concepts have been applied in existing ILP paradigms.

Chapter 3 introduces the multiscalar paradigm. It describes the basic philosophy of the paradigm, and compares and contrasts it with other processing paradigms such as the multiprocessor, VLIW, and superscalar paradigms. Chapter 3 also introduces the multiscalar processor, a possible implementation of the multiscalar paradigm.

Chapters 4-6 describe the different building blocks of the multiscalar processor. The description includes details of the instruction issue mechanism (distributed instruction caches), the distributed register file, the memory disambiguation unit, and the distributed data memory system. Chapter 4 describes the control unit, the execution units, and the instruction supply mechanism. Chapter 5 describes the distributed inter-operation communication mechanism, namely the replicated register files. Chapter 6 describes the distributed data memory system. It also describes how the multiscalar processor allows out-of-order execution of memory references before disambiguating their addresses, and how it detects any dependency violations brought about by out-of-order execution of memory references. The descriptions in chapters 5-6 are made as generic as possible so that the schemes developed in the context of the multiscalar processor can be used in other ILP processors too.

Chapter 7 presents the empirical results. It describes the methodology of experimentation, and the software tools developed specifically for studying the multiscalar processor. It presents the performance results of the multiscalar processor, and results showing that the distributed inter-operation communication mechanism in the multiscalar processor exploits localities of communication. Chapter 8 describes the role of the compiler, and the different ways in which the compiler can exploit the idiosyncrasies of the multiscalar paradigm to obtain more performance. Chapter 9 concludes the thesis by pointing out the broad implications of this dissertation.

The ARB mechanism proposed in chapter 6 is very general, and can be used in several execution models. The appendix demonstrates how the ARB can be used in superscalar processors, statically scheduled processors (such as VLIW processors), and small-scale multiprocessors.

Chapter 2. ILP Processing

What is the state of the art in ILP processing?

ILP processing involves a combination of software and hardware techniques to improve performance by executing multiple operations in parallel. This chapter describes the fundamental concepts related to ILP processing, and explores how these concepts have been applied in existing ILP paradigms. This chapter is organized in four sections. Section 2.1 describes the nature of parallelism in programs. Section 2.2 describes the different ways of specifying instruction level parallelism. Section 2.3 describes the fundamental steps involved in ILP processing, including the issues involved in extracting parallelism by software and by hardware. Section 2.4 reviews the state of the art in ILP processing, and provides a critique of the existing ILP paradigms. Section 2.5 recaps the main observations and concludes the chapter.

2.1. Program Execution Basics

From an abstract point of view, a program consists of computation operations and communication arcs, operating under control constraints. Computation operations operate on data values and produce new data values, which are communicated to other computation operations. The control constraints decide which parts of the program are executed, and how many times each part is executed. The basic idea behind ILP processing is to use multiple hardware resources to process multiple computation operations as well as multiple communication arcs in parallel so as to reduce the overall execution time. To better understand the nature of ILP in a program, we need to know the constraints on ILP and the ways in which they affect ILP processing.

2.1.1. Constraints on ILP

Data dependencies and control dependencies are the two fundamental constraints on ILP processing. Data dependencies relate to the communication arcs in the program; a data dependency is present when the result of one computation operation is needed by another operation. Data dependencies are true dependencies, and set a theoretical limit on the ILP that can be extracted from a program. A control dependency exists between two operations if the execution of one operation can prevent the execution of the other. Control dependencies manifest by way of conditionals in the program.

Because a control dependency can prevent the execution of other operations, it can introduce or eliminate data dependencies when a program is executed. Data dependencies can in turn affect control decisions that depend on them. This might lead one to suspect that control dependencies and data dependencies are indeed two sides of the same coin.

Control dependencies can be converted to data dependencies and vice versa

A data dependency between two operations can be trivially converted to a control dependency by specifying an execution ordering for the two computation operations. Techniques for converting control dependencies to data dependencies (e.g. if-conversion) are described in section 2.3.2.1.

The effect of data dependencies on parallelism get modulated based on the manner in which communication is handled. Depending on the way computation operations are mapped onto a processing structure, the communication arcs (data dependencies) get ‘‘stretched’’ or ‘‘shrunk’’, causing changes in the communication costs, because it takes more time to send values over a distance.

2.1.2. Program Representations

Irrespective of the programming language in which a program is written, for purposes of parallelism extraction, the compiler uses different representations for denoting the computations, the communication, and the controls embedded in a program. We shall briefly discuss the important types of representations. One of them is the *data flow graph (DFG)*, which is a directed graph whose nodes denote computation operations and whose arcs denote the communication of values, which define the necessary sequencing constraints. Control constraints are represented by selective routing of data among the nodes, using *switch* and *merge* nodes.

Another way of representing the attributes of a program is the *control flow graph (CFG)*, which is a directed graph whose nodes denote basic blocks (straight-line codes with no branches into or out of the middle of the block) and whose arcs denote the flow of control between them. Within a basic block, control is assumed to flow sequentially. Figure 2.1(i) shows an example CFG containing four basic blocks, marked BB1-BB4; the CFG contains a unique entry node called START and a unique exit node called STOP. In a control flow graph, a node *X* is said to be *post-dominated* by a node *Y* if every directed path from *X* to STOP (not including *X*) contains *Y* [51].

A third type of representation is the *program dependence graph (PDG)*, which explicitly represents the control dependencies and data dependencies for every operation in a program [51, 76]. The control dependencies are represented by a subgraph called *control dependence graph (CDG)*, and the data dependencies are represented by another subgraph called *data dependence graph (DDG)*. Control dependencies are derived from the control flow graph by constructing its post-dominator tree [51]. Figure 2.1(ii) shows the CDG of the CFG shown in Figure 2.1(i). Notice that *control parallelism* exists between basic blocks 1 and 4, *i.e.*, these two blocks are control independent. Data dependencies are derived by performing a data flow analysis of the program [7].

2.2. Program Specification and Execution Models

ILP processing involves a sequence of steps, some of which need to be done by software/programmer and some of which need to be done by hardware. In between comes extraction of parallelism, which can be done either by the compiler or by the hardware or by both. The instruction set architecture (ISA) of an ILP processor is a function of where the boundary is drawn between

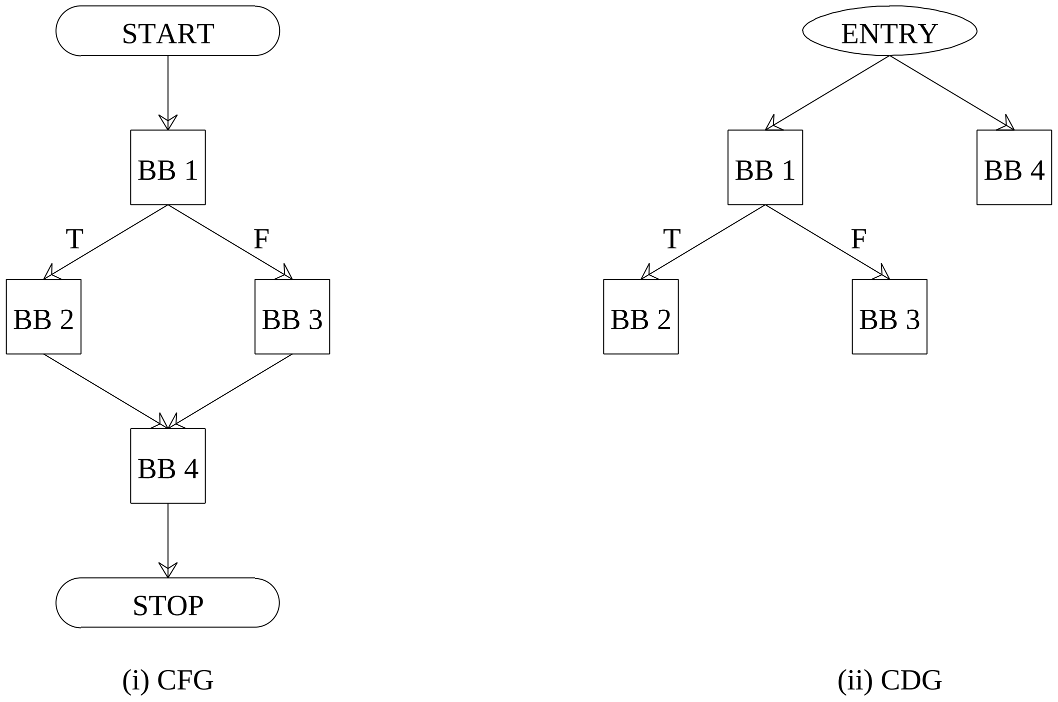


Figure 2.1: An Example Control Flow Graph (CFG) and its Control Dependence Graph (CDG)

what is done in software and what is done in hardware. The ISA is also a function of the manner in which information regarding parallelism is communicated by the compiler to the hardware through the program. In this regard, we consider two attributes to classify existing ILP architectures and their implementations. They are: (i) program specification order, and (ii) program execution order. These two attributes capture information about the extent to which static scheduling, dynamic scheduling, and combinations thereof are performed in different ILP processing paradigms.

2.2.1. Program Specification Order

2.2.1.1. Control-Driven Specification

The instructions in a program can imply either a static ordering or no ordering. When the instructions imply a static ordering, the corresponding ISA is said to be *control-driven*, and control is assumed to step through instructions one at a time. Communication is also control-driven; computation results do not directly proceed to the computation operations that need them, but are rather stored in a storage, for instance at the output of the functional units, in a set of registers, or in a set of memory locations. Control-driven architectures have the notion of a ‘‘present state’’ after each instruction is executed and the result(s) stored in the designated storage locations. The next instruction in the ordering is guaranteed to have its operands ready when control reaches it. Thus, all data dependencies are automatically specified by converting them to control dependencies! Generally, the implied program ordering is sequential ordering, with provisions to change the ordering by means of branch operations. Examples for control-driven specification are the VLIW, vector, and superscalar processors.

Control-driven specification allows communication to be specified by means of a shared storage structure. This has several implications. First, it gives rise to storage conflicts (*i.e.*, anti- and output dependencies). Second, execution of a conditional branch operation does not involve any routing of data, and involves only changing the value of the control sequencer. Third, it necessitates the construction of data access paths, computation operations to manipulate them, and algorithms to manage storage space, but therein lies the strength of static ordering. Static ordering can be used to introduce localities of communication, which can be exploited by hardware means to reduce the costs of communication, and to manage hardware resources properly. This has the effect of combining several adjacent computation operations of the program into one, because the communication arcs between them become very short when temporal locality is exploited by hardware means. Lastly, control-driven means ‘‘compiler-driven’’. The compiler can even analyze the entire program, and do some scheduling. The lesson learned from past experiences of the computer architecture community is to make use of whatever support and scheduling the compiler can do.

Multiple Flows of Control: The discussion carried out so far might suggest that in control-driven specification, the execution of one operation triggers the execution of exactly one other operation — *i.e.*, a single flow of control. However, this is not the case. In control-driven specification, control need not step through operations one at a time; each instruction could itself consist of many operations as in the VLIW paradigm. It is also possible to specify multiple flows of control in which the execution of an instruction triggers multiple instructions. An example for program specification involving multiple flows of control is the multiprocessing paradigm.

Parallelism Specification: In control-driven specification, the compiler can extract parallelism. The compiler-extracted parallelism can be specified in the program and conveyed to the execution hardware in many ways. One way is to use *data parallel* instructions (such as vector instructions) in which a single operation is specified on multiple data elements. Data parallel instructions are highly space-efficient; however, they are useful only for specifying the regular parallelism present in numeric programs, and are not of much use for specifying the irregular parallelism present in non-numeric programs. For specifying irregular parallelism, one can use *horizontal instructions* in which multiple operations are specified side by side in a single instruction. This is the specification used by the VLIW paradigm. Yet another type of parallelism specification is the one used by multiprocessors. In multiprocessors, the compiler specifies control-independent tasks, which can be executed by the hardware in parallel. Finally, in the case of the superscalar paradigm, the compiler can place independent instructions adjacent to each other in order that the superscalar hardware scheduler encounters them simultaneously.

2.2.1.2. Data-Driven Specification

When a program does not imply any particular static ordering, it is called *data-driven* specification. In this specification, communication is specified directly in the program. Computation results directly proceed to the computation operations that need them, and data are directly held inside instructions, instead of being stored elsewhere. In data-driven specification, control dependencies are handled by switch nodes and merge nodes, which route data conditionally. Thus, all control dependencies are automatically specified by converting them to data dependencies!

The perceived advantage of data-driven specification is that it helps to express the maximum amount of parallelism present in an application. Ironically, therein lies its weakness! Data-driven specification, in an attempt to express the maximum amount of parallelism to the hardware, refrains from doing any static scheduling, and pushes the scheduling task entirely to the hardware. Lack of a static ordering has other ramifications as well. Data-driven specification suffers from the inability to express critical sections and imperative operations that are essential for the efficient execution of operating system functions, such as resource management [36, 114]. Furthermore, in data-driven specification, data tokens have to be routed through control-decision points such as switch nodes and merge nodes, and often get collected at these control-decision points, waiting for the appropriate control decisions to be taken.

Data-Driven Specification with Control-Driven Tasks: In an attempt to capture the benefits of control-driven specification, researchers have proposed using control-driven specification at the lower level with data-driven specification at the higher level [77, 113]. In this specification, the dataflow graph is partitioned into different regions, each of which is scheduled independently at compile time. Within a region, control flows sequentially, whereas the execution of a region is triggered by inter-region data availability.

2.2.2. Program Execution Order

Program execution order is the ordering adopted by the execution hardware. If the hardware follows an execution order given by the program control flow, then it is called *control-driven execution*. If the hardware follows an execution order based on data availability, then it is called *data-driven execution*. In control-driven execution, a computation operation is triggered (*i.e.*, becomes ready for execution) when the control sequencer reaches that operation. Examples for control-driven execution are the traditional von Neumann, VLIW, and vector processors. In data-driven execution, a computation operation is triggered when all of its input operands have arrived. Examples for data-driven execution are the dataflow machines. Although existing paradigms that use data-driven specification have also used data-driven execution, this need not be the case. It is theoretically possible to have a hardware that takes a data-driven specification, converts it into control-driven specification and executes it in a control-driven manner.

Data-driven execution offers a lot of opportunity to execute multiple operations in parallel, because it introduces no artificial control restraints on parallel processing. However, the costs of freedom take their toll, which manifest as a host of overheads, rendering data-driven execution performance-ineffective. In an attempt to extract parallelism at run time, then, the data-driven hardware gets bogged down in a truly global scheduling every cycle. A significant communication overhead also accrues, as data-driven execution does not attempt to exploit localities of communication. Furthermore, data-driven execution needs special mechanisms to detect operand availability, to match results with needy operations, and to enable the execution of ready operations in an asynchronous manner. Finally, data-driven execution performs random scheduling; if many operations are ready for execution, then based on resource availability, some operations are selected in random for execution, without any consideration for the critical path. This is because the hardware cannot see what happens beyond the next cycle when no static ordering has been presented to it.

Control-driven execution definitely restrains parallelism. The advantage of control-driven execution is that after an instruction is executed, there is no need to search for the next instruction to be executed, which helps to reduce the hardware complexity significantly. Further, it has the capability to exploit localities of communication. In an attempt to combine the good aspects of control-driven execution and data-driven execution, processing paradigms have been proposed to combine control-driven execution and data-driven execution. Prime examples are the superscalar processors, which do data-driven execution within a window of operations obtained by control-driven fetching of instructions. Multiprocessors also use a combination of control-driven execution and data-driven execution, but in a different manner. In a multiprocessor, each processor performs control-driven execution, but inter-processor communication and synchronization are performed in a data-driven manner (with some hardware support).

2.3. Fundamentals of ILP Processing

Converting a high-level language program into one that a machine can execute involves taking several decisions at various levels. Parallelism exploitation involves additional decisions on top of this. The fundamental aspect in ILP processing is:

> Given a program graph with control and data constraints, arrive at a good execution schedule in which multiple computation operations are executed in a cycle as allowed by the resource constraints in the machine.

Arriving at a good schedule involves manipulations on the program graph, taking into consideration several aspects such as the ISA and the resources in the machine. Since there can only be a finite amount of fast storage (such as registers) for temporarily storing the intermediate computation values, the values have to be either consumed immediately or stored away into some form of backup storage (such as main memory), creating additional communication arcs. Thus, the challenge in ILP processing is not only to identify a large number of independent operations to be executed every cycle from a large block of computation operations having intricate control dependencies and data dependencies, but also reduce the inter-operation communication costs and the costs of storing temporary results. A good paradigm should not only attempt to increase the number of operations executed in parallel, but also decrease the inter-operation communication costs by reducing the communication distances and the temporary storing away of values, thereby allowing the hardware to be expanded as allowed by technology improvements in hardware and software.

Optimal scheduling (under finite resource constraints) is an NP-complete problem [32], necessitating the use of heuristics to take decisions. Although programmers can ease scheduling by expressing some of the parallelism present in programs by using a non-standard high-level language (HLL), the major scheduling decisions have to be taken by the compiler, by the hardware, or by both of them. There are different trade-offs in taking the decisions at programming time, at compile time, and at run time, as illustrated in Table 2.1.

A program's input (which can affect scheduling decisions) are available only at run-time when the program is executed, leaving compilers to work with conservative assumptions while taking scheduling decisions. Run-time deviations from the compile-time assumptions render the quality of

Table 2.1: Trade-offs in Taking Decisions at Programming Time, Compile Time, and Run Time

Taking Decisions at		
Programming Time	Compile Time	Run Time
No compiler complexity. No hardware complexity.	No hardware complexity.	Introduces hardware complexity, with potential cycle time increase. The benefit should more than make up for this increase in cycle time
Only macro-level analysis. Difficult to reason about parallelism, specific ISAs, & specific implementations.	Global analysis possible.	Only tunnel vision possible, with no increase in hardware complexity.
Conservative due to lack of run-time information.	Conservative due to lack of run-time information. Difficult to guarantee independence.	Has complete run-time information within the ‘‘tunnel’’, and can make well-informed decisions within the ‘‘tunnel’’.
Tailored for a specific. HLL program	HLL compatible. Tailored for specific ISA.	HLL compatible. Object code compatible.

the compiler-generated schedule poor, and increase the program execution time significantly. On the other hand, any scheduling decisions taken by the hardware could increase the hardware complexity, and hence the machine cycle time, making it practical for the hardware to analyze only small portions of the program at a time. Different ILP processing paradigms differ in the extent to which scheduling decisions are taken by the compiler or by the hardware.

In this section, we explore the different steps involved in ILP processing. To explore the full possibilities of what can be done by the compiler and what can be done by the hardware, this discussion assumes a combination of control-driven specification and data-driven execution. The sequence of processes involved in ILP processing can be grouped under the four steps listed below:
(i) *Expressing* ILP (done by the programmer),
(ii) *Extracting* ILP by software,
(iii) *Extracting* ILP by hardware, and
(iv) *Exploiting* ILP (done by the hardware).

In our opinion, these four steps are generally orthogonal to each other, although hardware mechanisms for extracting ILP (step 3) and those for exploiting ILP (step 4) are often integrated in hardware implementations. We shall look into each of the four steps in more detail.

2.3.1. Expressing ILP

The programmer can choose to express the parallelism present in an application by using appropriate algorithms, programming languages, and data structures. This is particularly useful for expressing the control parallelism present in an application. For a given application, some algorithms have more inherent parallelism than others. Similarly, some of the control dependencies in a program are an artifact of the programming paradigm used; dataflow languages such as Id [17] are tailored to express the parallelism present in an application, and do not introduce artificial control dependencies. One example where data structures help in expressing parallelism is the use of arrays instead of linked lists. Information about data parallelism can also be expressed by the programmer, by means of assertions. In many situations, the programmer may not be able to reason about and express all the parallelism in an application, in which case the compiler can attempt to expose some of the hidden parallelism, particularly the control parallelism in the program. These are described in detail in section 2.3.2.1.

2.3.2. Extracting ILP by Software

Extraction of ILP can be performed by software and by hardware. The motivation for using software to extract ILP is to keep the hardware simpler, and therefore faster. The motivation for using hardware to extract ILP (c.f. Section 2.3.3) is to extract that parallelism which can be detected only at run time. A central premise of this thesis is that these two methods are not mutually exclusive, and can both be used in conjunction to extract as much parallelism as possible.

There are three fundamental steps in extracting ILP from a program:

(1) Establish a window of operations.

(2) Determine and minimize dependencies between operations in this window.

(3) Schedule operations.

Below, we explain each of these steps in more detail.

2.3.2.1. Establishing a Window of Operations

The first step in extracting ILP from a program at compile time is to establish a path or a subgraph in the program's CFG, called an *operation window*. The two important criteria in establishing the operation window are that the window should be both large and accurate. Small windows tend to have small amounts of parallelism as discussed in section 1.2. Control dependencies caused by statically unpredictable conditional branches are the major hurdle in establishing a large and accurate static window. To overcome this, compilers typically analyze both paths of a conditional branch or do a prediction as to which direction the branch is most likely to go. Because an important component of most window-establishment schemes is the accurate prediction of conditional branches, a considerable amount of research has gone into better branch prediction techniques. Initial static prediction schemes were based on branch opcodes, and were not accurate. Now, static prediction schemes are much more sophisticated, and use profile information or heuristics to take decisions [28, 75, 98, 118, 160].

In addition to branch prediction, the compiler uses several other techniques to overcome the effects of control dependencies. Some of these techniques are if-conversion, loop unrolling, loop peeling, loop conditioning, loop exchange [12], function inlining [28, 85], replacing a set of IF-THEN statements by a jump table [129], and even changing data structures. All these techniques modify the CFG of the program, mostly by reducing the number of control decision points in the CFG. We shall review some of these schemes in terms of the type of modifications done to the CFG and how the modifications are incorporated.

Loop Unrolling: Loop unrolling is a technique used to enlarge the body of (typically) small loops. The key idea is to combine multiple iterations of the loop into a single iteration, and decrease the loop iterations proportionately, in order to permit better loop body pipelining and decrease the overhead of branch execution and induction variable update.

Loop Peeling: Loop peeling is a technique used to reduce dependencies when there is a control flow into the middle of a loop from outside the loop. If there is a control flow into the middle of a loop, then that control flow is effectively into the first iteration of the loop. Loop peeling involves peeling off this first iteration out of the loop by duplicating the loop body, and reducing the loop iteration count by one. Now, any dependencies that were carried into the loop through the previous control flow (into the loop) belong entirely to the peeled iteration, potentially allowing more static scheduling to be done.

Function Inlining: The essence of function inlining is to remove from the CFG of a program control flow alterations caused by function calls and returns. The procedure for doing inlining is to take a program's call graph and replace calls to the leaf functions (especially the ones that are called frequently) by the functions' body, and then repeating the process. Function inlining provides an enlarged window for code optimization, register allocation, and code scheduling. Most current compilers do function inlining as an optimization.

If-Conversion: A control dependency from operation $\mathbf{I_1}$ to operation $\mathbf{I_2}$ can be converted to a data dependency by changing $\mathbf{I_2}$ to a *conditional execution* operation whose execution is data dependent on a *guard*. The guard for a conditional execution operation is a Boolean expression that represents the conditions under which the operation is executed, and can be stored in a general purpose register or a condition code register. Thus, instead of the two operations

$\mathbf{I_1}$: **if (condition) branch to** $\mathbf{I_2}$

$\mathbf{I_2}$: **computation operation**

we can have,

$\mathbf{I_1}$: **guard** $\leftarrow$ **condition is true**

$\mathbf{I_2}$: **if (guard) computation operation**

This conversion process is called if-conversion [13, 14, 21, 35, 71, 129]. Because the guard is an input operand to $\mathbf{I_2}$, the relationship between $\mathbf{I_1}$ and $\mathbf{I_2}$ has now been converted from a control dependency to a data dependency. One potential problem with if-conversion, which prevents its pervasive application in a program, is that the if-converted instructions need to be executed irrespective of whether their execution is required or not.

2.3.2.2. Determining and Minimizing Dependencies

Once a window of operations has been established, the next step is to determine the data dependencies between the operations in the window, which exist through (pseudo)registers and memory locations. If register allocation has already been performed, then this step involves determining the register storage dependencies (anti- and output dependencies) as well.

Static Memory Address Disambiguation: Static memory address disambiguation is the process of determining if two memory references (at least one of which is a store) could ever point to the same memory location in any valid execution of the program. Static disambiguation is a hard task as memory addresses could correspond to pointer variables, whose values might change at run time. Two memory references may be dependent in one instance of program execution and not dependent in another instance, and static disambiguation has to consider all possible executions of the program. Various techniques have been proposed to do static disambiguation of memory references involving arrays [11, 19, 20, 49, 105, 163]. These techniques involve the use of conventional flow analyses of reaching definitions to derive symbolic expressions for array indexes. The symbolic expressions contain compile-time constants, loop-invariants, and induction variables, as well as variables whose values cannot be derived at compile time. To disambiguate two references, first their symbolic expressions are checked for equality. If the expressions are equal, then it means that the two references always conflict. If not, the two symbolic expressions are symbolically equated and the resulting diophantine equation is solved. If there are (integer) solutions to the equation, then it means that the two references *might* point to the same memory location during program execution, and a potential hazard has to be assumed. Otherwise, the two references are guaranteed not to conflict. For arbitrary multi-dimensional arrays and complex array subscripts, unfortunately, these tests can often be too conservative; several techniques have been proposed to produce exact dependence relations for certain subclasses of multi-dimensional arrays [62, 97]. Good static memory disambiguation is fundamental to the success of any parallelizing/vectorizing/optimizing compiler [14, 49, 54, 86, 94]. Existing static disambiguation techniques are limited to single procedures, and do not perform much of inter-procedural analysis. Moreover, existing static techniques are suitable only for memory references involving array subscripts, and not suitable for those references involving data types such as unions and pointers. Hence they are useful mainly for a class of scientific programs. Ordinary non-numerical applications, however, abound with unions and pointers [30, 67, 70, 92]. Although recent work has suggested that some compile-time analysis could be done with suitable annotations from the programmer [49, 68, 72], it appears that accurate static disambiguation for arbitrary programs written in arbitrary languages with pointers is unlikely without considerable advances in compiler technology.

Once the dependencies in the window are determined, the dependencies can be minimized by techniques such as software register renaming (if register allocation has been performed), induction variable expansion, and accumulator variable expansion. A description of some of these techniques is given below.

Software Register Renaming: Reuse of storage names (variables by the programmer and registers by the compiler) introduces artificial anti- and output dependencies, and restricts the static scheduler's opportunities for reordering operations. For instance, consider the code sequence given below.

$$\begin{aligned} I_1&:\ R1 = R1 + R2 \\ I_2&:\ R3 = R1 + R4 \\ I_3&:\ R1 = R4 \end{aligned}$$

In this code sequence, there is an anti-dependency between I_1 and I_3 and between I_2 and I_3 because of the reuse of register **R1**. Similarly, there is an output dependency between I_1 and I_3 through register **R1**. Because of these dependencies, I_3 cannot be executed concurrently or before I_2 or I_1. Many of these artificial dependencies can be eliminated with software register renaming. The idea behind software register renaming is to use a unique architectural register for each assignment in the window, in similar spirit to static single assignment [38]. In the above code, by replacing the assignment to register **R1** by an assignment to register **R5**, both the anti- and output dependencies are eliminated, allowing I_3 to be scheduled before I_2 and I_1 as shown below.

$$\begin{aligned} I_3&:\ R5 = R4 \\ I_1&:\ R1 = R1 + R2 \\ I_2&:\ R3 = R1 + R4 \end{aligned}$$

Induction Variable Expansion: Induction variables, used within loops to index through loop iterations and arrays, can cause anti-, output, and flow dependencies between different iterations of a loop. Induction variable expansion is a technique to reduce the effects of such dependencies caused by induction variables. The main idea is to eliminate re-assignments of the induction variable within the window, by giving each re-assignment of the induction variable a new induction variable name, thereby eliminating all dependencies due to multiple assignments. Of course, the newly introduced induction variables have to be properly initialized. Consider the loop shown in Figure 2.2(i). Figure 2.2(ii) shows this loop unrolled twice. The statements in the unrolled loop cannot be reordered because of the dependencies through the induction variable `i`. However, we can use two different induction variables, `i0` and `i1`, to redress the effects of multiple assignments to the induction variable, and reassign the induction variables as shown in Figure 2.2(iii).

```
i = 0
while (i < 100)
{
    A[i] += 10
    i++
}
```

(i)

```
i = 0
while (i < 100)
{
    A[i] += 10
    i++
    A[i] += 10
    i++
}
```

(ii)

```
i0 = 0;  i1 = 1
while (i0 < 100)
{
    A[i0] += 10;  A[i1] += 10
    i0 += 2;  i1 += 2
}
```

(iii)

Figure 2.2. Example Showing the Benefit of Induction Variable Expansion

(i) An Example Loop; (ii) The Loop Unrolled Twice;

(iii) Induction Variable Expansion

2.3.2.3. Scheduling Operations

Once an operation window is established, and the register dependencies and memory dependencies in the window are determined and minimized, the next step is to move independent operations up in the CFG, and schedule them in parallel with other operations so that they can be initiated and executed earlier than they would be in a sequential execution. If a static scheduler uses a basic block as the operation window, then the scheduling is called basic block scheduling. If the scheduler uses multiple basic blocks as the operation window, then the scheduling is called global scheduling. Basic block schedulers are simpler than global schedulers, as they do not deal with control dependencies; however, their use for extracting parallelism is limited. Global scheduling is more useful, as they consider large operation windows. Several global scheduling techniques have been developed over the years to establish large static windows and to carry out static code motions in the windows. These include: trace scheduling [53], percolation scheduling [57, 107], superblock scheduling [28], software pipelining [45, 89, 102, 103, 161], perfect pipelining [8, 9], boosting [141-143], and sentinel scheduling [95].

Trace Scheduling: The key idea of trace scheduling is to reduce the execution time along the more frequently executed paths, possibly by increasing the execution time in the less frequently executed paths. Originally developed for microcode compaction [53], trace scheduling later found application in ILP processing [34, 49]. Its methodology in the context of ILP processing is as follows: The compiler forms the operation window by selecting from an acyclic part of the CFG the most likely path, called *trace*, that will be taken at run time. The compiler typically uses profile-based estimates of conditional branch outcomes to make judicious decisions in selecting the traces. There may be conditional branches out of the middle of the trace and branches into the middle of the trace from outside. However, the trace is treated and scheduled as if there were no control dependencies within the trace; special compensation codes are inserted on the off-trace branch edges to offset any changes that could affect program correctness. Then the next likely path is selected and scheduled, and the process is continued until the entire program is scheduled. Nested loops are scheduled by first applying trace scheduling on the inner loop body, and then treating the inner loop as a single node within the enclosing outer loop. Trace scheduling is a powerful technique, which is very useful for numeric programs in which there are a few most likely executed paths. In non-numeric programs, however, many conditional branches are statically difficult to predict, let alone have a high probability of branching in any one direction.

Percolation Scheduling: Percolation scheduling considers a subgraph of the CFG (such as an entire function) as a window, using a system of semantics-preserving transformations for moving operations between adjacent blocks [107]. Repeated application of the transformations allows operations to ‘‘percolate’’ towards the top of the program, irrespective of their starting point. A complete set of semantics-preserving transformations for moving operations between adjacent blocks is described in [107]. Ebcioglu and Nakatani describe an enhanced implementation of percolation scheduling for VLIW machines having conditional evaluation capabilities [47]. In their implementation of percolation scheduling, a code motion across a large number of basic blocks is allowed only if each of the pair-wise transformations is beneficial.

Superblock Scheduling: Superblock scheduling [27, 28] is a variant of trace scheduling. A superblock is a trace with a unique entry point and one or more exit points, and is the operation window used by the compiler to extract parallelism. Superblocks are formed by identifying traces using profile information, and then using tail duplication to eliminate any control entries into the middle of the trace. (Tail duplication means making a copy of the trace from the side entrance to the end of the trace, and redirecting the incoming control flow path to that copy.) In order to generate large traces (preferably along the most frequently executed paths of the program), techniques such as branch target expansion, loop peeling, and loop unrolling are used. Once a superblock is formed, the anti-, output, and flow dependencies within the superblock are reduced by techniques such as software register renaming, induction variable expansion, and accumulator variable expansion. Finally, scheduling is performed within the superblock by constructing the dependence graph and then doing list scheduling. In order to reduce the effect of control dependencies, operations are speculatively moved above conditional branches.

Hyperblock Scheduling: In hyperblock scheduling, the operation window is a hyperblock. A hyperblock is an enhancement on superblock [96]. A hyperblock is a set of predicated basic blocks in which control may enter only from the top, but may exit from one or more points. The difference between a hyperblock and a superblock is that a superblock contains instructions from only instructions from one path of control, whereas a hyperblock contains instructions from multiple paths of control. If-conversion is used to convert control dependencies within the hyperblock to data dependencies. The predicated instructions are reordered without consideration to the availability of their predicates. The compiler assumes architectural support to guarantee correct execution.

Software Pipelining: The static scheduling techniques described so far deal mostly with operation windows involving acyclic codes. Software pipelining is a static technique for scheduling windows involving loops. The principle behind software pipelining is to overlap or pipeline different iterations of the loop body. The methodology is to do loop unrolling and scheduling of successive iterations until a repeating pattern is detected in the schedule. The repeating pattern can be re-rolled to yield a loop whose body is the repeating schedule. The resulting software pipeline schedule consists of a prologue, a kernel, and an epilogue, and its execution is similar to that of a sequence of chained vector instructions. Figure 2.3 illustrates how software pipelining is applied to a loop. Figure 2.3(i) shows an example loop; Figure 2.3(ii) shows the loop unrolled three times. Scheduling is done on the unrolled loop, and the re-rolled loop is shown in Figure 2.3(iii). Different techniques have been proposed to do software pipelining: perfect pipelining [10], enhanced pipeline scheduling [47], GURPR* [149], modulo scheduling [48, 124], and polycyclic scheduling [125].

Boosting: Boosting is a technique for statically specifying speculative execution [141-143]. Conceptually, boosting converts control dependencies into data dependencies using a technique similar to if-conversion, and then executes the if-converted operations in a speculative manner before their guards are available. (The guard of a boosted operation includes information about the branches on which the boosted operation is control dependent and the statically predicted direction of each of these branches.) Extra buffering is provided in the processor to hold the effect of speculative

```
    R0 = 0
L:  Load R1
    R2 = R1 + R3
    Store R2
    R0 = R0 + 1
    BLT R0, 100, L
```

(i)

```
    R0 = 0
L:  Load R1
    R2 = R1 + R3
    Store R2
    Load R1
    R2 = R1 + R3
    Store R2
    Load R1
    R2 = R1 + R3
    Store R2
    R0 = R0 + 3
    BLT R0, 100, L
```

(ii)

```
    R0 = 2
    Load R1                                   } Prologue
    R2 = R1 + R3;  Load R1
L:  Store R2;  R2 = R1 + R3;  Load R1;  R0 = R0 + 1;  BLT R0, 99, L
    Store R2;  R2 = R1 + R3                   } Epilogue
    Store R2
```

(iii)

Figure 2.3. Example Illustrating Software Pipelining

(i) An Example Loop; (ii) The Loop Unrolled Thrice; (iii) The Software-Pipelined Loop

operations. When the guard of a speculatively executed operation becomes available, the hardware checks if the operation's execution was required. If the execution was required, the non-speculative state of the machine is updated with the buffered effects of that operation's execution. If the operation should not have been executed, the hardware simply discards the state and side-effects of that operation's execution. Boosting provides the compiler with additional opportunity for reordering operations, while making the hardware responsible for ensuring that the effects of speculatively-executed operations do not affect the correctness of program execution when the compiler is incorrect in its speculation.

Advantages of Static Extraction of ILP: The singular advantage of using the compiler to extract ILP is that the compiler can do a global and much more thorough analysis of the program than is possible by the hardware. It can even consider the entire program as a single window, and do global scheduling in this window. Furthermore, extraction of ILP by software allows the hardware to be simpler. In any case, it is a good idea to use the compiler to extract whatever parallelism it can extract, and to do whatever scheduling it can to match the parallelism to the hardware model.

Limitations of Static Extraction of ILP: Static extraction of ILP has its limitations. The main limitation is the extent to which static extraction can be done for non-numeric programs in the midst of a conglomeration of ambiguous memory dependencies and data-dependent conditional branches. The inflexibility in moving ambiguous memory operations can pose severe restrictions on static code motion in non-numeric programs. Realizing this, researchers have proposed schemes that allow ambiguous references to be statically reordered, with checks made at run time to determine if any dependencies are violated by the static code motions [31, 108, 132]. Ambiguous references that are statically reordered are called *statically unresolved references*. A limitation of this scheme, however, is that the run-time checks need extra code and in some schemes associative compare of store addresses with preceding load addresses in the active window. Another issue of concern in static extraction of ILP is *code explosion*. An issue, probably of less concern nowadays, is that any extraction of parallelism done at compile time is architectural, and hence may be tailored to a specific architecture or implementation. This is not a major concern, as specific compilers have become an integral part of any new architecture or implementation.

2.3.3. Extracting ILP by Hardware

Given a program with a particular static ordering, the hardware can change the order and execute instructions concurrently or even out-of-order in order to extract additional parallelism, so long as the data dependencies and control dependencies in the program are honored. There is a price paid in doing this run-time scheduling, however. The price is the complexity it introduces to the hardware, which could lead to potential increases in cycle time. For hardware scheduling to be effective, any increase in cycle time should be offset by the additional parallelism extracted at run time. When the hardware extracts ILP, the same 3 steps mentioned in section 2.3.2 are employed. However, instead of doing the 3 steps in sequence, the hardware usually overlaps the steps, and performs all of them in each clock cycle.

2.3.3.1. Establishing a Window of Instructions

To extract large amounts of ILP at run time, the hardware has to establish a large window of instructions. It typically does that by fetching a fixed number of instructions every cycle, and collecting these instructions in a hardware window structure. The main hurdles in creating a large dynamic window are control dependencies, introduced by conditional branches. To overcome these hurdles, the hardware usually performs *speculative fetching* of instructions. With speculative fetching, rather than waiting for the outcome of a conditional branch to be determined, the branch outcome is predicted, and operations from the predicted path are entered into the window for execution. Dynamic prediction techniques have significantly evolved over the years [93, 112, 133, 166]. Although the accuracies of contemporary dynamic branch prediction techniques are fairly high, averaging 95% for the SPEC '89 non-numeric programs [166], the accuracy of a large window obtained through n independent branch predictions in a row is only $(0.95)^n$ on the average, and is therefore poor even for moderate values of n. Notice that this problem is an inherent limitation of following a single line of control. The new ILP paradigm that we describe in this thesis breaks this restriction by following multiple flows of control.

2.3.3.2. Determining and Minimizing Dependencies

In parallel to establishing the window, the hardware also determines the different types (flow, anti-, and output) of register and memory dependencies between the instructions in the window. Register dependencies are comparatively easy to determine as they require only the comparison of the source and destination operand specifiers of the operations. Determining memory dependencies is harder, and is described below.

Dynamic Memory Address Disambiguation: To determine the memory dependencies in the established window, memory references must be disambiguated. Disambiguating two memory references at run time means determining if the two references point to the same memory location or not. In processors that perform dynamic extraction of parallelism, dynamic disambiguation involves comparing the addresses of all loads and stores in the active window; existing hardware mechanisms invariably perform this comparison by means of associative searches, which becomes extremely complex for large windows. Chapter 6 further addresses the issues involved in dynamic disambiguation. Over the years, different techniques have been proposed for performing dynamic disambiguation [15, 115, 139]; these are also described in detail in chapter 6. Another important aspect in connection with dynamic disambiguation is that existing schemes execute memory references after performing disambiguation. The problem with this approach is that a memory reference has to wait until the addresses of all the preceding stores become known. If worst-case assumptions are made about possible memory hazards and memory references (especially the loads) are made to wait until the addresses of all preceding stores are known, much of the code reordering opportunities are inhibited, and performance may be affected badly. A solution to this problem is to execute memory references as and when they are ready — before they are properly disambiguated — as *dynamically unresolved references*. When a store is executed, a check can be made to determine if any dynamically unresolved reference has been executed to the same address from later in the sequential instruction stream. If any such incorrect reference is found, the processor can recover from the incorrect execution, possibly with the help of recovery facilities already provided for doing speculative execution of code.

After determining the register and memory dependencies in the window, the next focus is on reducing the anti- and output dependencies (storage conflicts) in the window, in order to facilitate aggressive reordering of instructions. The natural hardware solution to reduce such storage conflicts is to provide more physical storage, and use some dynamic renaming scheme to map from the limited architectural storage to the not-so-limited physical storage. An example for this technique is register renaming, which we shall look at in more detail.

Hardware Register Renaming: Storage conflicts occur very frequently with registers, because they are limited in number, and serve as the hub for inter-operation communication. The effect of these storage conflicts becomes very severe if the compiler has attempted to keep as many values in as few registers as possible, because the execution order assumed by a compiler is different from the one the hardware attempts to create. A hardware solution to decrease such storage conflicts is to provide additional physical registers, which are then dynamically allocated by hardware register renaming techniques [84]. With hardware register renaming, typically a free physical register is allocated for every assignment to a register in the window, much like the way software register renaming allocates architectural registers. Several techniques have been proposed to implement hardware register renaming [66, 73, 116, 147, 153].

2.3.3.3. Scheduling Instructions

In parallel to establishing a window and enforcing the register and memory dependencies, the hardware performs scheduling of ready-to-execute instructions. Instructions that are speculatively fetched from beyond unresolved branches are executed speculatively, *i.e.*, before determining that their execution is needed. The hardware support for speculative execution consists of extra buffering in the processor, which holds the effects of speculatively executed instructions. When a conditional branch is resolved, if the earlier prediction was correct, all speculative instructions that are directly control dependent on the branch are committed. If the prediction was incorrect, then the results of speculatively executed instructions are discarded, and instructions are fetched and executed from the correct path. Several dynamic techniques have been proposed to implement speculative execution coupled with precise state recovery [73, 74, 115, 116, 138, 147, 154].

Hardware schedulers often use simplistic heuristics to choose from the instructions that are ready for execution. This is because any sophistication of the instruction scheduler directly impacts the hardware complexity. A number of dynamic scheduling techniques have been proposed: CDC 6600's scoreboard [151], Tomasulo's algorithm [153], decoupled execution [134], register update unit (RUU) [147], dispatch stack [43, 44], deferred-scheduling, register-renaming instruction shelf (DRIS) [123], etc. Below, we briefly review each of these schemes; Johnson provides a more detailed treatment of these schemes [78].

Scoreboard: Scoreboard is a hardware technique for dynamic scheduling, originally used in the CDC 6600 [151], which issued instructions one at a time. However, the scoreboard technique can be extended to issue multiple instructions per cycle. The hardware features in the scoreboard technique are: a *buffer* associated with each functional unit, and a *scoreboard*. The buffer is for holding an instruction that is waiting for operands, and the buffers collectively form the instruction window. The scoreboard is a centralized control structure that keeps track of the source registers and destination registers of each instruction in the window, and their dependencies. The scoreboard algorithm operates as follows. Instructions are fetched and issued as long as the relevant functional units are available. If the instruction's operands are not available, the instruction waits in the buffer associated with its functional unit. Instruction issue stops when a functional unit is needed by multiple instructions. The scoreboard enforces the data dependencies between the instructions in the window by controlling the correct routing of data between functional units and registers. When all the source registers of a pending instruction have valid data, the scoreboard determines this condition, and issues the command to forward data from the register file to the correct functional unit, and to start the instruction's execution. Similarly, when a functional unit finishes execution, the scoreboard issues the control signal to route the result from the functional unit to the register file. The limitations of the scoreboard technique are: (i) instruction issue stops when a functional unit is needed by multiple instructions (this was a restriction of the scoreboard described in [151], and not an inherent limitation of the scoreboard technique), (ii) instruction issue stops when there is an output dependency, (iii) the centralized scoreboard becomes very complex when the window becomes large.

Tomasulo's Algorithm: Tomasulo's out-of-order issue algorithm was first presented for the floating point unit of the IBM 360/91 [15], which also issued instructions one at a time. The hardware features for carrying out Tomasulo's algorithm are: a *busy bit* and a *tag field* associated with each register, a set of *reservation stations* associated with each functional unit, and a *common data bus (CDB)*. The busy bit of a register is to indicate if the register contains the latest value or not. The tag field of a register is to indicate the reservation station from which the register will receive its latest instance if its busy bit is set. The reservation stations are for holding instructions that are waiting for operands, and they have fields for storing operands, their tags, and busy bits. The reservation stations collectively form the instruction window. The CDB connects the outputs of all functional units to all registers and reservation stations. Tomasulo's algorithm operates as follows. When an instruction is decoded, it is issued to a reservation station associated with its functional unit. The busy bits of its source registers and their tag fields or value fields (depending on whether the register is busy or not) are also forwarded to the reservation station. The tag field of the destination register is updated with the tag corresponding to the reservation station. By copying the source registers' tags to the reservation stations and by updating the destination register's old tag with a new tag, hardware register renaming is performed. Once at a reservation station, an instruction can get its operands by monitoring the CDB and by comparing the tags of the results appearing in the CDB. Instructions that have all their operands available are ready to be executed. If multiple instructions are ready at a functional unit, one of them is selected for execution. Tomasulo's algorithm is very ambitious in terms of code reordering opportunities because it removes all anti- and output dependencies in the active window. However, the algorithm can be expensive to implement for large windows because of the associative comparison hardware for tag matching. Its register renaming scheme also introduces imprecise interrupts, because the register file is allowed to be updated out-of-order. Tomasulo's algorithm does distribute the dynamic scheduling logic throughout the execution unit; however, the results of all computation operations have to be distributed throughout the execution unit using the CDB, resulting in ‘‘global’’ communication.

RUU: The register update unit (RUU) technique proposed by Sohi and Vajapeyam [144, 147], is an enhancement of Tomasulo's algorithm, and guarantees precise interrupts. The hardware features for carrying out this scheme are: two *counter* fields associated with each register, and a *register update unit*. The counter fields of a register are for keeping track of the number of outstanding instances destined for the register and the latest instance destined for the register. The RUU is a centralized queue-like structure that stores the decoded instructions of the window, keeps track of their dependencies, and related information. Instead of having separate reservation stations for each functional unit, the reservation stations are combined together in the RUU, which helps to improve the utilization of reservation stations. The RUU algorithm operates as follows. In every cycle, the RUU performs four distinct tasks in parallel: (i) accepts a decoded instruction from the issue logic, and places it at the tail, (ii) issues a ready instruction (if any) to a functional unit, (iii) monitors the result bus for matching tags, and (iv) commits the instruction at the head (by writing its results to the register file) if it has completed. The RUU holds decoded instructions in the sequential order in which they appear in the program, and these instructions wait in the RUU until their operands become ready. The results of completed instructions are not routed directly to the register file; instead they are routed only to the RUU to allow waiting instructions to pick up their data. Results are forwarded to the register file in sequential order, as and when instructions are committed from the RUU.

Dispatch Stack: The dispatch stack is a technique proposed by Acosta *et al* [4, 154]. The hardware features for carrying out this scheme are: two *dependency count* fields associated with each register, and a *dispatch stack*. The dependency count fields of a register are for holding 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). The dispatch stack is a centralized structure that keeps track of the source registers and destination registers of each instruction in the window. The dispatch stack technique works as follows. When an instruction is decoded, it is forwarded to the dispatch stack, where the dependency count fields of its source and destination registers are updated by comparing the register specifiers with those of the instructions already in the dispatch stack. An instruction is ready for execution (subject to functional unit availability) when both of its dependency counts are zero. As instructions complete, the dependency counts are decremented based on the source and destination register specifiers of completing instructions. The disadvantages of this scheme are: (i) storage conflicts are not overcome, and (ii) the centralized dispatch stack with its associated comparators ($5iw + 5fw$ comparators for a dispatch stack of size w in an i-issue machine with f functional units) becomes very complex when the window becomes large.

DRIS: The deferred-scheduling, register-renaming instruction shelf (DRIS) technique proposed by Popescu *et al* [123], is a variant of the RUU technique described earlier, and performs superscalar issue coupled with speculative execution. The hardware features for carrying out this scheme are: two *counter* fields associated with each register, and a *register update unit*. The DRIS is a centralized queue-like structure that stores the decoded instructions of the window, keeps track of their dependencies, and related information. The DRIS algorithm operates as follows. In every cycle, five distinct tasks are done in parallel: (i) the issue logic issues multiple instructions to the DRIS after determining their data dependencies to previous instructions by searching the DRIS, (ii) the scheduler examines all instructions in the DRIS in parallel, identifying those whose operands are all available, (iii) the execution units execute the instructions selected by the scheduler, (iv) the instruction results are written into the appropriate DRIS entries, and (v) among the completed instructions, the oldest ones that do not have the interrupt bit set are retired. Just like the RUU, the DRIS also holds the instructions in their sequential order. The DRIS hardware scheduler has been implemented in the Lightning processor [123], which is a superscalar implementation of the Sparc architecture.

Decoupled Execution: Decoupled architectures exploit the ILP between the memory access operations and the core computation operations in a program. The hardware features for carrying out decoupled execution are: two processors called the *access processor* and the *execute processor*, and *architectural queues* coupling them. Decoupled execution works as follows: The instruction stream is split into access and execute instructions either by the compiler or by the hardware. The access processor executes the access instructions and buffers the data into the architectural queues. The execute processor executes the core computation operations. The access processor is allowed to slip ahead of the execute processor, and to prefetch data from memory ahead of when it is needed by the latter, thereby decoupling the memory access operations from the core computations. Several decoupled architectures have been proposed, and some have been built. The noted ones are the MAP-200 [33], the DAE [136], the PIPE [63], the ZS-1 [139], and the WM architecture [165]. A drawback in decoupled architectures is AP-EP code unbalance. When there is unbalance, the sustained instruction issue rate is low because of poor utilization. Further, if the access and execute processors are single-issue processors, then the issue rate can never exceed two operations per cycle.

Advantages of Dynamic Extraction of ILP: The major advantage in doing (further) extraction of ILP at run-time is that the hardware can utilize the information that is available only at run time to extract the ILP that could not be extracted at compile time. In particular, the hardware can resolve ambiguous memory dependencies, which cannot be resolved at compile time, and use that information to make more informed decisions in extracting ILP. The schedule developed at run time is also better adapted to run-time uncertainities such as cache misses and memory bank conflicts.

Limitations of Dynamic Extraction of ILP: Although dynamic scheduling with a large centralized window has the potential to extract large amounts of ILP, a realistic implementation of a wide-issue (say a 16-issue) processor with a fast clock is not likely to be possible because of its complexity. A major reason has to do with the hardware required to parse a number of instructions every cycle. The hardware required to extract independent instructions from a large centralized window and to enforce data dependencies typically involves wide associative searches, and is non-trivial. While this hardware is tolerable for 2-issue and 4-issue processors, its complexity increases rapidly as the issue width is increased. The major issues of concern for wide-issue processors include: (i) the ability to create accurate windows of perhaps 100s of instructions, needed to sustain significant levels of ILP, (ii) elaborate mechanisms to enforce dependencies between instructions in the window, (iii) possibly wide associative searches in the window for detecting independent instructions, and (iv) possibly centralized or serial resources for disambiguating memory references at run time.

2.3.4. Exploiting Parallelism

Finally, adequate machine parallelism is needed to execute multiple operations in parallel, irrespective of whether parallelism is extracted at run time or not. Machine parallelism determines the number of operations that can be fetched in a cycle and executed concurrently. Needless to say, the processor should have adequate functional units to perform many computations in parallel. Another important aspect is the resources to handle communication. If there are altogether n dynamic communication arcs, and a maximum of m communication arcs can be executed simultaneously, then the program execution time involves a minimum of n/m communication steps, no matter how many parallel resources are thrown at for computations. If the critical resource in the system has an average bandwidth of B_c, then the sustained issue rate is upper bounded by $\frac{B_c}{U_c C_c}$ operations per cycle, where U_c is the average number of times the critical resource is used by an operation, and C_c is the average number of cycles the resource is used each time. In order to provide adequate bandwidth, all critical resources should be decentralized.

Key to decentralization of resources is exploiting localities of communication

To decentralize hardware resources, it is not sufficient to physically distribute the hardware. What is more important is to restrict communication to local areas within the processor as much as possible. The key to doing this is to exploit the localities of communication by hardware means.

Below, we expound the need for some of the important hardware features for exploiting ILP.

Instruction Supply Mechanism: Irrespective of the ILP processing paradigm used, the hardware has to fetch and decode several operations every cycle — the sustained issue rate can never exceed the number of operations fetched and decoded per cycle. Decoding many operations in parallel is a complex task; a good processing paradigm should facilitate decentralization of the instruction supply mechanism.

Inter-Operation Communication: A major issue that the hardware needs to address has to do with the communication/synchronization between the multiple operations that are simultaneously active. Modern processors invariably use a load-store architecture, with the register file serving as the hub for communication between all functional units. To sustain an *average* issue rate of I operations per cycle in a load/store architecture with dyadic ALU operations, the register file has to provide a bandwidth of *at least* $2I$ reads and I writes per cycle. The common way of providing this bandwidth is to use a multi-ported register file. Whereas a few multi-ported register files have been built, for example the register files for the Cydra 5 [127], the SIMP processor [101], Intel's iWarp [79], and the XIMD processor [162], centralized, multi-ported register files do not appear to be a good long-term solution, as its design becomes very complex for large values of I. Therefore, decentralization of the inter-operation communication mechanism is essential for future ILP processors. Ideally, the decentralized mechanism should retain the elegance of the architectural register file, with its easy and familiar compilation model, but at the same time provide high bandwidth with a decentralized realization.

Support for Speculative Execution: When a processor performs speculative execution, the effects of speculatively executed operations can be allowed to update the machine's architectural state only when they are guaranteed to commit; otherwise the machine's architectural state will be lost, complicating the recovery procedures in times of incorrect branch prediction. Nevertheless, succeeding operations (from speculatively executed code) often do require access to the new uncommitted machine state, and not the architectural state. Thus, the inter-operation communication mechanism and the data memory system in a speculative execution machine have to provide some means of forwarding uncommitted values to subsequent operations, and the architectural state has to be updated (if need be) in the order given by the sequential semantics of the program.

Data Memory System: Regardless of the ILP processing paradigm used, a low latency and high bandwidth data memory system is crucial to support the data bandwidth demands of ILP processing. The data bandwidth demands of ILP processors are much higher than those of single-issue processors because at least the same number of memory references, and possibly more if speculative execution is performed, are executed in fewer clock cycles.

Because hardware mechanisms for exploiting ILP are generally designed to support specific processing paradigms, the details of the mechanisms are intimately tied to technology factors and the implementation details of the processing paradigm used. We shall not discuss these hardware mechanisms in this thesis; instead, we move on to a critical examination of the existing ILP processing paradigms.

2.4. A Critique on Existing ILP Processing Paradigms

2.4.1. The Dataflow Paradigm

The dataflow paradigm uses data-driven specification and data-driven execution in an attempt to exploit the maximum amount of parallelism. Massive parallelism can be exploited if the data-driven execution mechanism could be implemented cost-effectively with low instruction execution overhead. The pure dataflow paradigm is very general, and exploits parallelism at the operation level. It uses an unconventional programming paradigm in the form of a dataflow graph (data-driven specification) to express the maximum amount of parallelism present in an application. The dataflow execution engine conceptually considers the entire dataflow graph to be a single window at run time (data-driven execution). The major advantage of the dataflow paradigm is that its data-driven specification is the most powerful mechanism for expressing parallelism. A disadvantage of dataflow machines is that they require special (dataflow) languages to harness their potential. While conceptually imperative languages, such as C, could be executed on a dataflow machine, we are unaware how this could be done in a performance-competitive manner. Another major disadvantage is that although the model allows the maximum number of computation operations to be triggered simultaneously, it ignores the inter-operation communication aspects completely. A large performance penalty is incurred due to the ‘‘stretching’’ of the communication arcs of the dataflow graph, which manifests as associative searches through the entire program using matching hardware. The recent advent of more restricted forms of dataflow architectures [37, 77, 109, 113, 114] bear testimony to this fact.

The dataflow paradigm has evolved significantly over the years, from the early tagged token store architectures [39, 158], to the more recent explicit token store machines such as the Monsoon [113], P-RISC [109], TAM (Threaded Abstract Machine) [37], and *T [110], and other hybrid machines [24, 77]. In the Monsoon architecture, the associative waiting-matching store of these machines was replaced by an explicitly managed, directly addressed token store. In the P-RISC architecture, the complex dataflow instructions (each with the capability to do low-level synchronization) were split into separate synchronization, arithmetic, and fork/control instructions, eliminating the necessity of presence bits on the token store. Furthermore, the compiler was permitted to assemble instructions into longer threads, replacing some of the dataflow synchronization with conventional program counter-based synchronization. The TAM architecture also allowed the compiler to introduce additional instruction sequencing constraints, making explicit the notion of a thread. Thus, we can see that the dataflow architectures are fast moving towards the conventional architectures, in which the compilers perform a significant part of instruction scheduling.

2.4.2. The Systolic Paradigm and Other Special Purpose Paradigms

Several special purpose paradigms have been proposed over the years to exploit localities of communication in some way or other. The noted one among them is the systolic paradigm [88], which is excellent in exploiting ILP in numeric applications such as signal processing and computer vision [16, 23]. Some specialized computers even hard-wire FFTs and other important algorithms to give tremendous performance for signal processing applications. The applicability of such special purpose processing paradigms to non-numeric applications, having complex control and data flow patterns, is not clear. This is evidenced by the recent introduction of the iWarp machine [23, 117], where support is provided for both systolic communication and coarse-grain communication.

2.4.3. The Vector Paradigm

The earliest paradigm for ILP processing was the vector paradigm. In fact, much of the compiler work to extract parallelism was driven by vector machines. The vector paradigm uses control-driven specification (with multiple operations per instruction) and control-driven execution. Thus, all the exploited ILP is extracted at compile-time[1]. A vectorizing compiler establishes a large static window, typically by considering multiple iterations of the inner loops in a program. Independent operations from subsequent iterations are moved up, and vector instructions are used to express the fact that operations from later iterations could be executed earlier than they would be initiated in sequential execution. For instance, the loop

```
for (i = 0; i < 100; i++)
     A[i] = B[i] + 10;
```

can be expressed by a single vector instruction

```
A[0:99] = B[0:99] + 10
```

provided arrays A and B do not overlap in memory. The execution of a vector instruction initiates the execution of operations from the later iterations earlier than they would be initiated in sequential

[1] Restricted data-driven execution can occur if the hardware allows vector instructions to start and finish out-of-order, in which case dynamic extraction of ILP also takes place.

execution. Examples for vector machines are CDC Star-100 [150], TI ASC [150], Cray machines [2, 3, 130, 131], CDC Cyber-205 [1], and Fujitsu VP-200 [99]. Vector machines are a good example for exploiting localities of communication; the *chaining* mechanism directly forwards a result from the producer to the consumer without an intermediate storage [131]. Vector machines perform remarkably well in executing codes that fit the vector paradigm [157]. Unfortunately, very few non-numeric programs can be vectorized with existing compiler technology, rendering vector machines as an inappropriate way of exploiting ILP in such programs. This is especially the case for programs that use pointers. As mentioned earlier, we do not expect static analysis techniques for pointers, and consequently the vectorization of programs having pointers, to change dramatically in the next several years.

2.4.4. The VLIW Paradigm

The VLIW paradigm also uses control-driven specification (with multiple operations per instruction) and control-driven execution. All the exploited ILP is again extracted at compile time. The compiler takes (a path or subgraph of) the control flow graph (CFG), and extracts ILP from that by moving data- and control-independent operations up in the CFG. However, instead of using a compact, vector instruction to specify parallelism, the VLIW paradigm uses a large instruction word to express the multiple operations to be issued and executed in a cycle. Thus, the VLIW paradigm is more general than the vector paradigm. At run-time, the VLIW hardware executes all operations in a VLIW instruction in lock-step. After the completion of each VLIW instruction, the paradigm guarantees that the results of all operations in the instruction are made available to all operations of the subsequent instruction in the next clock cycle. Thus, the VLIW hardware needs to provide a crossbar-like inter-operation communication mechanism (e.g. multi-ported register file). The pros and cons of the VLIW approach have been discussed in detail in the literature [29, 34, 49, 53, 54, 56, 125, 146]. With suitable compiler technology, a VLIW machine, typified by the Multiflow Trace series [34] and the Cydrome Cydra-5 [127], is able to perform well for vectorizable and non-vectorizable numeric codes [100, 126]. Because no dynamic scheduling is performed, the hardware is somewhat simpler.

The disadvantages of VLIW include: (i) code compatibility, (ii) inability to extract large amount of parallelism because of conservative static decisions, (iii) inability to adapt to run time uncertainities, (iv) sophisticated compiler support, and (v) need for centralized resources such as large, multi-ported register files, crossbars for connecting the computation units, and wide paths to the issue unit.

Traditional VLIW execution hardware followed a single flow of control. The ability to follow multiple flows of control is very important if significant levels of ILP are to be sustained for ordinary codes. Serious thought is being given to expanding the abilities of the VLIW model to allow the execution of multiple branches per cycle; several proposals have been made [46, 83, 100, 162]. The XIMD approach allows a VLIW processor to adapt to the varying parallelism in an application, as well as to execute multiple instruction streams [162]. The processor coupling approach is similar in spirit to XIMD, but it allows dynamic interleaving of multiple threads to tolerate long memory latencies [83]. In the IBM VLIW research project [46], the form of a VLIW instruction is expanded to that of a directed binary tree, with provision to express conditional operations that depend on multiple condition codes. The effectiveness of these approaches for applications that are not amenable to static analysis techniques (to extract parallelism) is not clear.

2.4.5. The Superscalar Paradigm

An alternate paradigm that was developed at the same time as the VLIW paradigm (early 80's) was the superscalar paradigm [80, 101, 111, 116]. The superscalar paradigm attempts to bring together the good aspects of control-driven specification and data-driven execution, by doing data-driven execution within a window of instructions established by control-driven fetching along a single flow of control. Parallelism may be extracted at compile time and at run time. To extract parallelism at run time, the hardware traverses the CFG by doing in parallel the following: (i) establish a path through the CFG, (ii) fetch instructions from the established path into a centralized hardware window structure, (iii) construct a dataflow graph of the instructions in the window, and (iv) traverse the constructed dataflow graph. For establishing a path through the CFG, the superscalar processor typically uses branch prediction. For instance, in the CFG of Figure 2.1(i), the hardware could establish path BB1-BB2-BB4 or path BB1-BB3-BB4 based on the prediction made at the branch terminating BB1. If an established path is later found to be incorrect at some node on the path, the entire path after that node is discarded even though some parts of the discarded path may still be a part of the actual path to be taken (e.g., BB4 in Figure 2.1(i)). The fundamental problem is that the superscalar hardware has no access to the higher-level control flow of the program; all the control flow information that it has is what is obtained in a step-by-step manner by the decoding of instructions.

There have been several proposals for superscalar machines in the 1980s [4, 33, 87, 101, 119, 120, 134, 135, 155, 156, 164]; notable ones include the IBM Cheetah, America and RIOS projects which culminated in the IBM RS/6000 [66, 111], decoupled architectures, which resulted in the Astronautics ZS-1 [137, 139], and HPS [73, 116]. Advantages of the superscalar approach include object code compatibility and the ability to make run-time decisions and adapt to run-time uncertainities (for example variable memory latencies encountered in cache-based systems), potentially extracting more parallelism. These advantages have led to the adoption of small-issue superscalar processors as the paradigm of choice for several modern microprocessors [35, 40, 123]. The major limitations of the superscalar paradigm include: (i) inability to create accurate window sizes of perhaps 100s of instructions, when following a single line of control, (ii) inability to handle large windows with small cycle times, (iii) extremely wide instruction cache, and other data paths/switches, (iv) elaborate mechanisms to forward data from a producer to a consumer, (v) possibly wide associative searches, and (vi) possibly centralized (or serial if not centralized) resources, for example, resources for disambiguating memory references, amongst others.

2.4.6. The Multiprocessing Paradigm

Although not belonging to the ILP processing world, it is worthwhile to discuss the multiprocessing paradigm. In this paradigm, the compiler transforms the CFG of a program into a *multiple-flow CFG*, where the multiple flows are usually at a coarse granularity. (It is also possible for the programmer to directly write a parallel program.) Wherever a data dependency exists between the multiple control flows, the compiler inserts an explicit synchronization operation. The multiprocessing hardware takes the multiple-flow CFG, and executes the multiple control flows using multiple processors. Each processor executes the assigned task sequentially; collectively the multiprocessor executes several instructions per cycle, one from each control flow.

2.5. Observations and Conclusions

This chapter provided a review of the state of the art in ILP processing, in terms of both the hardware and the software issues. Table 2.2 compares the existing ILP paradigms, in terms of what is done by the software and what is done by the hardware.

Table 2.2: Comparison of Existing ILP Paradigms

	Dataflow	Vector	VLIW	Superscalar	Multiprocessor
Expressing ILP (Special languages)	Critical	Not critical	Not critical	Not critical	Not critical
Extracting ILP by software	No	Critical	Critical	Not critical	Critical
Extracting ILP by hardware	Critical	Not critical	No	Critical	Not critical
Exploiting ILP	Critical	Critical	Critical	Critical	Critical
Program spec. order	Data-driven	Control-driven	Control-driven	Control-driven	Control-driven & Data-driven
Program execution order	Data-driven	Control-driven	Control-driven	Control-driven & Data-driven	Control-driven & Data-driven
Parallelism specification	Not specified	Vector instr.	Horiz. instr.	Not specified	Independent tasks

Arriving at a good execution schedule involves many steps, each of which involves taking different decisions. There are different trade-offs in taking these decisions at programming time, compile time, and run time. Although the programmer can give some helpful hints about the parallelism present in an application, it is not advisable to delegate the entire responsibility of parallelism-finding to the programmer. Although the compiler can take quite a bit of scheduling decisions, quite often it finds it difficult to give *guarantees* of independence due to lack of run-time information. Finally, it is worthwhile to take scheduling decisions (or for that matter, any decisions) at run time only if the benefits are more than the detrimental effects the decision-making has on the hardware cycle time.

Chapter 3. The Multiscalar Paradigm

How to execute control- and data-dependent tasks in parallel?

We saw that existing ILP processing paradigms have an over-reliance on compile-time extraction or run-time extraction of ILP, and fail to exploit the localities of communication present in ordinary programs. In this chapter, we introduce a new processing paradigm — the *multiscalar* paradigm — which not only combines the best of both worlds in ILP extraction, but also exploits the localities of communication present in programs. Because of these and a host of other features, which we will study in this chapter, we believe that the multiscalar paradigm will become a cornerstone for future ILP processors. The name multiscalar is derived from the fact that the overall computing engine is a collection of scalar processors that cooperate in the execution of a sequential program. In the initial phases of this research, the multiscalar paradigm was called the *Expandable Split Window (ESW)* paradigm [58].

This chapter is organized in five sections. The first section describes our view of an ideal processing paradigm. Naturally, the attributes mentioned in section 3.1 had a significant impact on the development of the multiscalar paradigm and later became the driving force behind an implementation of the paradigm. Section 3.2 introduces the multiscalar paradigm, and explains its basic idea and working with the help of two examples. Section 3.3 describes the interesting and novel aspects of the multiscalar paradigm. Section 3.4 compares and contrasts the multiscalar paradigm with some of the existing processing paradigms such as the multiprocessor, superscalar, and VLIW paradigms. Section 3.5 introduces the multiscalar processor, our implementation of the multiscalar paradigm. Section 3.6 summarizes the chapter by drawing attention to the highlights of the multiscalar paradigm.

3.1. Ideal Processing Paradigm—The Goal

Before embarking on a discussion of the multiscalar paradigm, it is worth our while contemplating on the desired features that shaped its development. Ideally, these features should take into consideration the hardware and software technological developments that we expect to see in the next several years. We can categorize the features into those related to the software issues and those related to the hardware issues. In a sense, they represent the trade-offs we wish to make regarding what is to be done in software and what is to be done in hardware. First, let us look at the software issues. These issues can be classified under three attributes, namely practicality, parallelism, and versatility.

(i) *Practicality:* On the software side, by practicality we mean the ability to execute ordinary programs on the processor. The paradigm should not require the programmers to write programs in specific programming languages; instead, programmers should be given the freedom to write programs in ordinary, imperative languages such as C. The programmers should not be forced to spend too much effort finding the low-level parallelism in an application. In short, the paradigm should place no unnecessary burden on the programmers to carry out ILP processing.

(ii) *Versatility:* As far as possible, the high-level language programs should not be tailored for specific architectures and specific hardware implementations, so that the same high-level language program can be used for a wide variety of architectures and implementations. The programmer should not have to consider the number or logical connectivity of the processing elements in the computer system.

(iii) *Parallelism*: The compiler should extract the maximum amount of parallelism possible at compile time. The compiler could also convey additional information about the program, such as register dependencies and control flow information, to the hardware. These steps will not only simplify the hardware, but also allow it to concentrate more on extracting the parallelism that can be detected only at run time.

Now let us consider the desired features for the hardware. Interestingly, the desired hardware features can also be classified under the same three attributes, namely parallelism, practicality, and versatility.

(i) *Parallelism:* The hardware should extract the parallelism that could not be detected at compile time, and should exploit the maximum amount of parallelism possible. The hardware should be able to execute multiple flows of control.

(ii) *Practicality:* Here by practicality we mean realizability of the hardware. That is, the ILP paradigm should have attributes that facilitate commercial realization. A processor based on the paradigm should be implementable in technology that we expect to see in the next several years, and the hardware should be regular to facilitate implementation with clock speeds comparable to the clock speeds of contemporary single-issue processors, resulting in the highest performance processor of a given generation.

(iii) *Versatility:* The paradigm should facilitate hardware implementations with no centralized resources. Decentralization of resources is important to later expand the system (as allowed by technology improvements in hardware and software). These resources include the hardware for extracting ILP such as register and memory dependency enforcement and identification of independent operations, and the hardware for exploiting ILP such as instruction supply mechanism, inter-operation communication, and data memory system. The hardware implementation should be such that it provides an easy growth path from one generation of processors to the next, with minimum (hardware and software) effort. An easy hardware growth path implies the reuse of hardware components, as much as possible, from one generation to the next.

3.2. Multiscalar Paradigm—The Basic Idea

Realization of the software and hardware features described above has precisely been the driving force behind the development of the multiscalar paradigm. Bringing all of the above features together requires bringing together in a new manner the worlds of control-driven execution and data-driven execution, and combine the best of both worlds.

The basic idea of the multiscalar paradigm is to split the jobs of ILP extraction and exploitation amongst multiple processing elements. Each PE can be assigned a reasonably sized task, and parallelism can be exploited by overlapping the execution of multiple tasks. So far, it looks no different from a multiprocessor. But the difference — a key one indeed — is that the tasks being executed in parallel in the multiscalar paradigm can have both control and data dependencies between them. Whereas the multiprocessor takes control-independent portions (preferably data-independent as well) of the control flow graph (CFG) of a program, and assigns them to different processing elements, the multiscalar takes a sequential instruction stream, and assigns contiguous portions of it to different processing elements. The multiple processing elements, or execution units if you will, are connected together as a circular queue. Thus, the multiscalar paradigm traverses the CFG of a program as follows: take a subgraph (task) T from the CFG and assign it to the tail execution unit, advance the tail pointer by one execution unit, do a prediction as to where control is most likely to go after the execution of T, and assign a subgraph starting at that target to the next execution unit in the next cycle, and so on until the circular queue is full. The assigned tasks together encompass a contiguous portion of the dynamic trace. These tasks are executed in parallel, although the paradigm preserves logical sequentiality among the tasks. The units are connected as a circular queue to obtain a *sliding* or *continuous* big window (as opposed to a fixed window), a feature that allows more parallelism to be exploited [159]. When the execution of the subgraph at the head unit is over (determined by checking when control flows out of the task), the head pointer is advanced by one unit.

A task could be as simple as a basic block or part of a basic block. More complex tasks could be sequences of basic blocks, entire loops, or even entire function calls. *In its most general form, a task can be any subgraph of the control flow graph of the program being executed.* The motivation behind considering a subgraph as a task is to collapse several nodes of the CFG into a single node, as shown in Figure 3.1. Traversing the CFG in steps of subgraphs helps to tide over the problem of poor

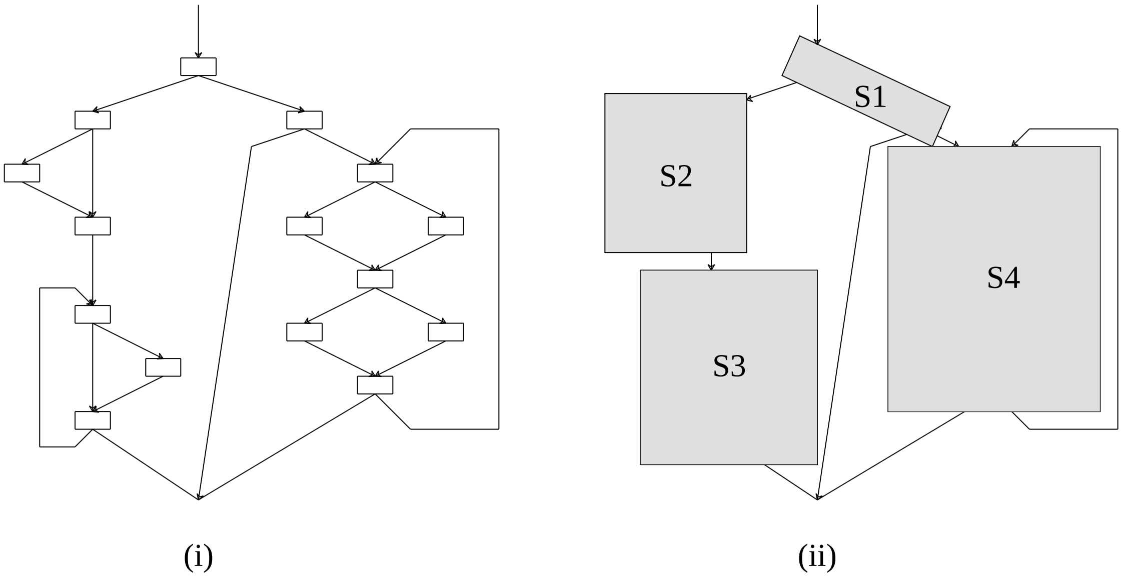


Figure 3.1: Forming Multiscalar Tasks from a CFG

(i) A Collection of Basic Blocks; (ii) A Collection of Tasks

predictability of some CFG nodes, by incorporating those nodes within subgraphs[2]. Parallelly executed tasks can have both control dependencies and data dependencies between them. The execution model within each task can be a simple, sequential processing paradigm, or more complicated paradigms such as a small-issue VLIW or superscalar paradigm.

Let us throw more light on multiscalar execution. The multiscalar paradigm executes multiple tasks in parallel, with distinct execution units. Each of these execution units can be a sequential,

---

[2] A distant, but interesting, analogy in physics is that of high wavelength waves getting less dispersed (than small wavelength waves) by particles, because they can bend over particles.

single-issue processor. Collectively, several instructions are executed per cycle, one from each task. Apart from any static code motions done by the compiler, by simultaneously executing instructions from multiple tasks, the multiscalar execution moves some instructions ‘‘up in time’’ within the overall dynamic window. That is, some instructions from later in the sequential instruction stream are initiated earlier in time, thereby exploiting parallelism, and decreasing the overall execution time. Notice that the compiler did not give any guarantee that these instructions are independent; the hardware determines the inter-task dependencies (possibly with additional information provided by the compiler), and determines the independent instructions. If a new task is assigned to a different execution unit each cycle, collectively the execution units establish a large dynamic window of instructions. If all active execution units execute instructions in parallel, overall the multiscalar processor could be executing multiple instructions per cycle.

3.2.1. Multiscalar Execution Examples

We shall illustrate the details of the working of the multiscalar paradigm with the help of examples. The examples are not meant to be exclusive but are meant to be illustrative. Let us start with a simple loop. Later, we shall go into a more complicated example.

Example 3.1: Consider the execution of the simple code fragment shown in Figure 3.2. Figures 3.2(i)-(iii) show the C code, its assembly code, and its control flow graph, respectively. The example is a simple loop with a data-dependent conditional branch in the loop body. The loop adds the number **b** to 100 elements of an array A, and sets an element to 1000 if it is greater than 1000. The CFG consists of three basic blocks BB1-BB3. This example is chosen for its simplicity. Whereas it does not illustrate some of the complexities of the control flow graphs that are generally encountered in practice, it does provide a background for discussing these complexities.

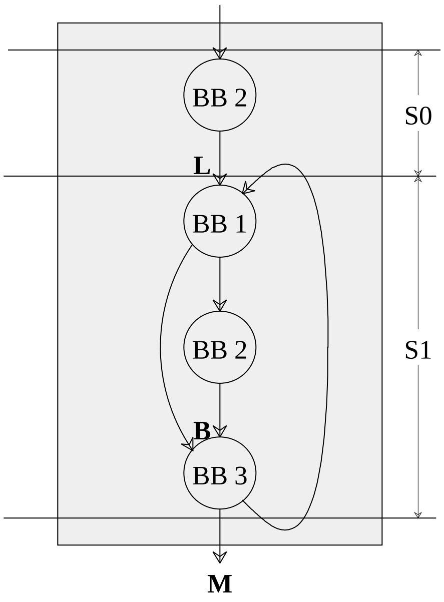
On inspection of the assembly code in Figure 3.2(ii), we can see that almost all the instructions of an iteration are data-dependent on previous instructions of the same iteration, and that there is very

```
for (i = 0; i < 100; i++)
{
        A[i] += 10;
        if (A[i] > 1000)
                A[i] = 1000;
}
```

(i) Example C Code

```
    R1  =  -1           ; initialize induction variable i
    R0  =  b            ; assign loop-invariant b to R0
L:  R1  =  R1 + 1       ; increment i
    R2  =  Mem[R1 + A]  ; load A[i]
    R2  =  R2 + R0      ; add b
    BLT  R2, 1000, B    ; branch to B if A[i] < 1000
    R2  =  1000         ; set A[i] to 1000
B:  Mem[R1 + A]  =  R2  ; store A[i]
    BLT  R1, 99, L      ; branch to L if i < 99
```

(ii) Assembly Code



(iii) Control Flow Graph

Figure 3.2: Example Code Fragment

little ILP in a single iteration of the loop. However, all the iterations are independent (except for the data dependencies through the loop induction variable `i` allocated in register R1) because each iteration operates on a different element of the array. Now, let us look at how the multiscalar paradigm executes this loop. First, the compiler demarcates the code such that the loop body is considered as a single task, marked S1 in Figure 3.2(iii). This task has two possible targets, L and M. At run time, the processor expands the loop into multiple dynamic tasks as shown in Figure 3.3, effectively establishing a large dynamic window of consecutive iterations. The large dynamic window encompasses a contiguous portion of the dynamic trace. The multiscalar paradigm executes these multiple tasks in parallel, with distinct execution units. Collectively, several instructions are executed per cycle, one from each task. For instance, consider the shaded portion in Figure 3.3, which refers to a particular time-frame (cycle). In that cycle, three instructions are executed from the three tasks.

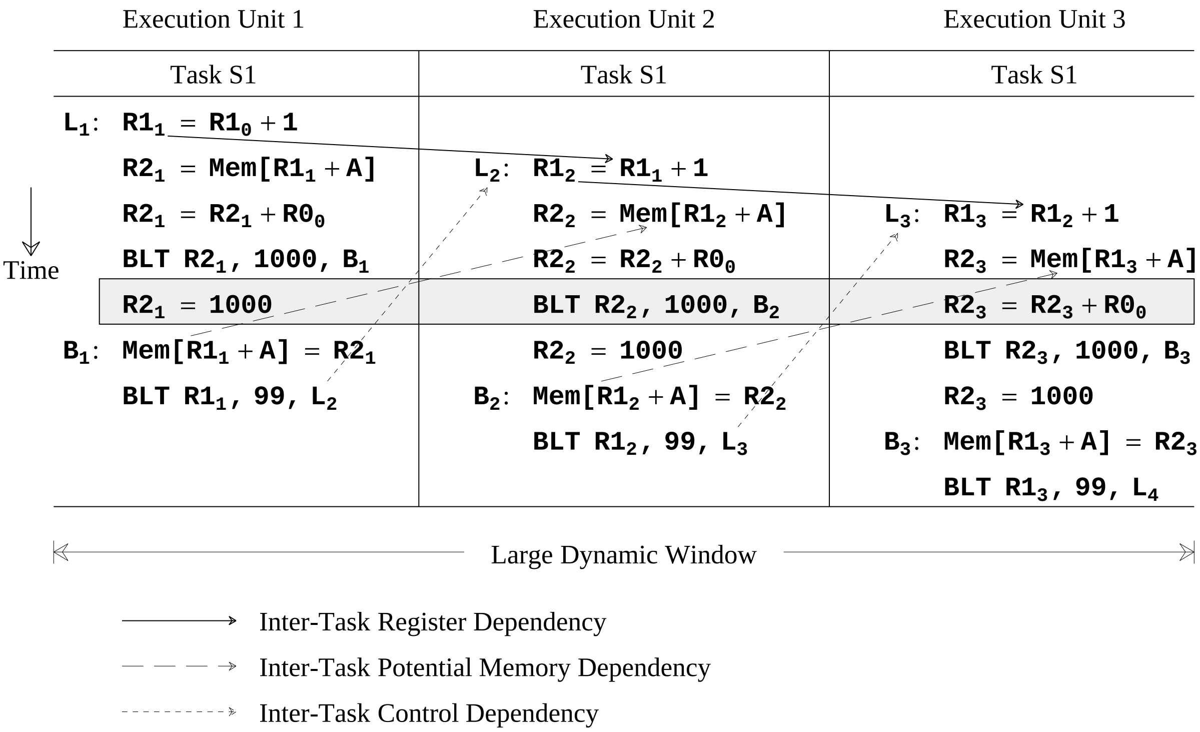


Figure 3.3. Multiscalar Execution of Assembly Code in Figure 3.2(ii)

Given the background experience assumed here, it would be coy not to recognize the reader's familiarity with software scheduling techniques such as loop unrolling and software pipelining. However, it cannot be emphasized too often that the multiscalar paradigm is far more general than loop unrolling and other similar techniques for redressing the effect of control dependencies. The structure of a multiscalar task can be as general as a connected subgraph of the control flow graph, and is far more general than a loop body. This is further clarified in Example 3.2. Before that, let us look into more detail how inter-task control dependencies and data dependencies are handled in the multiscalar paradigm.

Control Dependencies: We will first see how inter-task control dependencies are overcome. Once task 1 is dynamically assigned to execution unit 1, a prediction is made by the hardware (based on static or dynamic techniques) to determine the next task to which control will most likely flow after the execution of task 1. In this example, it determines that control is most likely to go back to label L, and so in the next cycle, another instance of the same task, namely task 2, is assigned to the next execution unit. This process is repeated. We call the type of prediction used by the multiscalar paradigm as *control flow prediction* [121]. In the multiscalar paradigm, the execution of all active tasks, except the first, is speculative in nature. The hardware provides facilities for recovery when it is determined that an incorrect control flow prediction has been made. It is important to note that among the two branches in an iteration of the above loop, the first branch, which has poor predictability, has been encompassed within the task so that the control flow prediction need not consider its targets at all while making the prediction. Only the targets of the second branch, which can be predicted with good accuracy, have been included in the task's targets. Thus the constraints introduced by control dependencies are overcome by doing speculative execution (along the control paths indicated by the light dotted arrows in Figure 3.3), but doing predictions at those points in the control flow graph that are easily predictable. This facilitates the multiscalar hardware in establishing accurate and large dynamic windows.

Register Data Dependencies: Next we will look at how inter-task register data dependencies are handled. These data dependencies are taken care of by forwarding the last update of each register in a task to the subsequent tasks, preferably as and when the last updates are generated. In Figure 3.3, the register instances produced in different tasks are shown with different subscripts, for example, $R1_1$, $R1_2$, and $R1_3$, and the inter-task register data dependencies are marked by solid arrows. As we can gather from Figure 3.3, the only register data dependencies that are carried across the tasks are the ones through register R1, which corresponds to the induction variable. Thus, although the instructions of a task are mostly sequentially dependent, the next task can start execution once the first instruction of a task has been executed (in this example), and its result forwarded to the next task.

Memory Data Dependencies: Now let us see how potential inter-task data dependencies through memory, occurring through loads and stores, are handled. These dependencies are marked by long dash arrows in Figure 3.3. In a sequential execution of the program, the load of the second iteration is performed after the store of the first iteration, and thus any potential data dependency is automatically taken care of. However, in the multiscalar paradigm, because the two iterations are executed in parallel, it is quite likely that the load of the second iteration may be ready to be executed earlier than the store of the first iteration. If a load is made to wait until all preceding stores are executed, then much of the code reordering opportunities are inhibited, and performance may be affected badly. The multiscalar paradigm cannot afford such a callous wait; so it allows memory references to be executed out-of-order, along with special hardware to check if the dynamic reordering of memory references produces any violation of dependencies. For this recovery, it is possible to use the same facility that is provided for recovery in times of incorrect control flow prediction. If the dynamic code motion rarely results in a violation of dependencies, significantly more parallelism can be exploited. This is a primary mechanism that we use for breaking the restriction due to ambiguous data dependencies, which cannot be resolved by static memory disambiguation techniques.

Example 3.2: Next we will look at a more complicated example, one that involves complex control flows. This example also highlights the advantage of encapsulating an entire loop (not just an iteration of a loop) within a task. Encapsulating an inner loop within a task may be advantageous if one or more of the following apply: (i) the loop body contains only a few instructions, (ii) the loop iterates only a few times, (iii) the iterations are data-dependent. A realistic implementation of the multiscalar hardware can be expected to assign only one new task every cycle, and therefore the sustained operation issue rate of the processor is limited by the average size of a task. If the body of an inner loop contains only a few instructions, then considering each iteration to be a task results in small tasks, and hence low issue rates. If an inner loop iterates only two or three times and an iteration is considered to be a task, then the control flow prediction accuracy will be very poor. Similarly, if the iterations are data-dependent, then there is little merit in assigning them to different execution units at run time. Consider the C code shown in Figure 3.4(i), taken from the *xlisp* program in the SPEC '92 benchmark suite. It consists of a doubly nested loop; the inner loop walks down a linked list, and its iterations are both control and data dependent. However, each activation of the inner loop is independent of the previous one. If the entire inner loop is encompassed within a single task, it is possible to execute many activations of the inner loop in parallel.

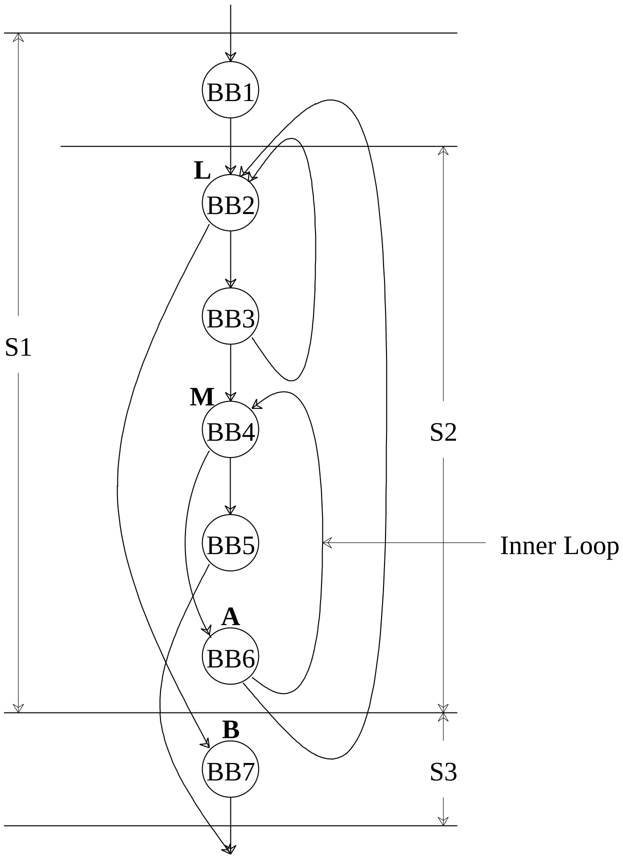
Now let us consider a multiscalar execution of this example code. The compiler has demarcated the CFG into three tasks, S1-S3, as shown in Figure 3.4(ii). The two important things to notice are: (i) tasks S1 and S2 overlap, and (ii) the entire inner loop is contained within task S2. Tasks S1 and S2 have the same three targets, namely labels L, B, and RETURN. Task S3 has one target, namely RETURN. Figure 3.5 shows a possible execution in the multiscalar paradigm. First task S1 is allocated to execution unit 1. In the next cycle, among S1's three targets, the most likely one (*i.e.*, L) is chosen, and the task starting at label L (*i.e.*, S2) is assigned to execution unit 2. Similarly, in the third cycle, the most likely target of S2 (label L) is predicted, and task S2 is again assigned to the next execution unit. Notice that the tasks being executed themselves contain the entire inner loop, as shown by the long dashed arrows in Figure 3.5.

```
for (fp = xlenv; fp; fp = cdr(fp))
{
        for (ep = car(fp); ep; ep = cdr(ep))
        {
                if (sym == car(car(ep)))
                        return (cdr(car(ep)));
        }
}
return (sym);
```

(i) C Code

```
    R1  =  xlenv - 8   ;
L:  R1  =  Mem[R1 + 8] ; load fp = xlenv or cdr(fp)
    BEQ  R1, 0, B      ; branch to B if fp is zero
    R2  =  Mem[R1 + 0] ; load ep = car(fp)
    BEQ  R2, 0, L      ; branch to L if ep is zero
M:  R3  =  Mem[R2 + 0] ; load car(ep)
    R4  =  Mem[R3 + 0] ; load car(car(ep))
    BNE  R0, R4, A     ; branch to A if sym ≠ car(car(ep))
    R5  =  Mem[R3 + 8] ; load cdr(car(ep)) as return value
    RET  R5            ; return R5
A:  R2  =  Mem[R2 + 8] ; load ep = cdr(ep)
    BNE  R2, 0, M      ; branch back to M if ep ≠ zero
    B  L               ; branch back to L
B:  R5  =  Mem[R0 + 8] ; load cdr(sym) as return value
    RET  R5            ; return R5
```

(ii) Assembly Code



(iii) Control Flow Graph

Figure 3.4: Example Code from Function Xlygetvalue in Xlisp

Execution Unit 1	Execution Unit 2	Execution Unit 3
Task S1	Task S2	Task S2
$R1_1 = \mathrm{xlenv\_addr} - 8$		
L_1: $R1_1 = Mem[R1_1 + 8]$		
$BEQ\ R1_1, 0, B_1$	L_2: $R1_2 = Mem[R1_1 + 8]$	
$R2_1 = Mem[R1_1 + 0]$	$BEQ\ R1_2, 0, B_2$	L_3: $R1_3 = Mem[R1_2 + 8]$
$BEQ\ R2_1, 0, L_2$	$R2_2 = Mem[R1_2 + 0]$	$BEQ\ R1_3, 0, B_3$
M_1: $R3_1 = Mem[R2_1 + 0]$	$BEQ\ R2_2, 0, L_3$	$R2_3 = Mem[R1_3 + 0]$
$R4_1 = Mem[R3_1 + 0]$	M_2: $R3_2 = Mem[R2_2 + 0]$	$BEQ\ R2_3, 0, L_4$
$BNE\ R0_0, R4_1, A_1$	$R4_2 = Mem[R3_2 + 0]$	M_2: $R3_3 = Mem[R2_3 + 0]$
$R5_1 = Mem[R3_1 + 8]$	$BNE\ R0_0, R4_2, A_2$	$R4_3 = Mem[R3_3 + 0]$
$RET\ R5_1$	$R5_2 = Mem[R3_2 + 8]$	$BNE\ R0_0, R4_3, A_3$
A_1: $R2_1 = Mem[R2_1 + 8]$	$RET\ R5_2$	$R5_3 = Mem[R3_3 + 8]$
$BNE\ R2_1, 0, M_1$	A_2: $R2_2 = Mem[R2_2 + 8]$	$RET\ R5_3$
$B\ L_2$	$BNE\ R2_2, 0, M_2$	A_2: $R2_3 = Mem[R2_3 + 8]$
	$B\ L_3$	$BNE\ R2_3, 0, M_3$
		$B\ L_4$

Time

Large Dynamic Window

Inter-Task Register Dependency

Intra-Task Loop

Inter-Task Control Dependency

Figure 3.5. Multiscalar Execution of Assembly Code in Figure 3.4(ii)

3.3. Interesting Aspects of the Multiscalar Paradigm

The astute reader would have realized by now that the multiscalar paradigm allows very flexible dynamic scheduling that could be assisted with software scheduling. The compiler has a big role to play in bringing to fruit the full capabilities of this paradigm. The compiler decides which parts of the CFG should be brought together as a task, and performs static scheduling within each task. The role of the compiler is discussed in great detail in chapter 9. Figure 3.6 gives a clear picture of where the multiscalar paradigm stands in terms of what is done by software and what is done by hardware.

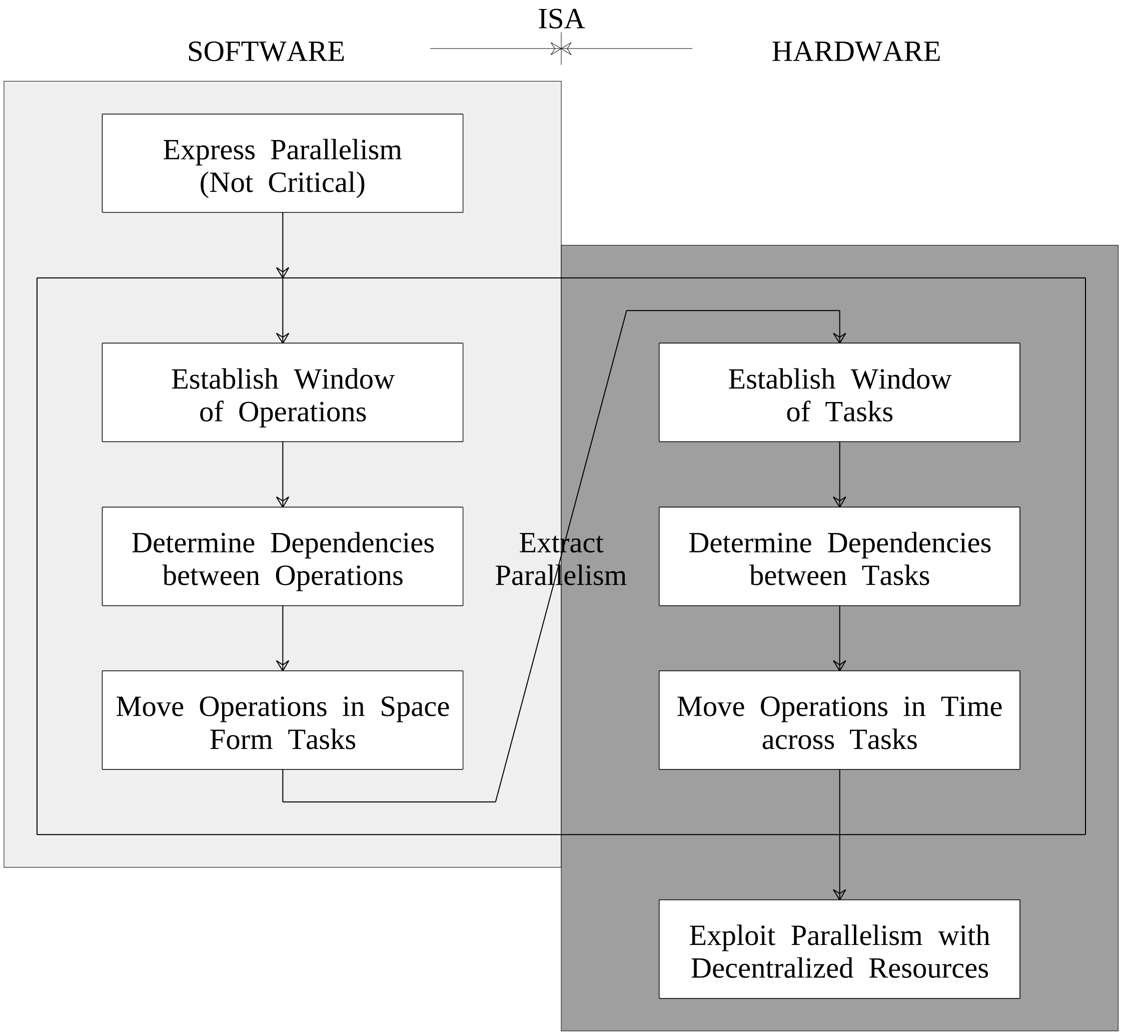


Figure 3.6: The Multiscalar Paradigm—What is done by Software and What is done by Hardware

The multiscalar paradigm is grounded on a good interplay between compile-time extraction of ILP and run-time extraction of ILP. Below, we describe the interesting aspects of the multiscalar paradigm.

3.3.1. Decentralization of Critical Resources

The next several chapters describe one possible implementation of the multiscalar paradigm. Without considering the details of the multiscalar implementation here, we can make one observation upon the strategy it employs for decentralizing the critical resources. By splitting the large dynamic window of instructions into smaller tasks (c.f. Figure 3.7), the complex task of searching a large window for independent instructions is split into two simpler subtasks: (i) independent searches (if need be) in smaller tasks, all of which can be done in parallel by separate execution units, and (ii) enforcement of control and data dependencies between the tasks. This allows the dynamic scheduling hardware to be divided into a two-level hierarchical structure — a distributed top-level unit that enforces dependencies between the tasks, and several independent lower-level units at the bottom level, each of which enforces dependencies within a task and identifies the independent instructions in that task. Each of these lower-level units can be a separate execution unit akin to a simple (possibly sequential) execution datapath. A direct outgrowth of the decentralization of critical resources is expandability of the hardware.

3.3.2. Execution of Multiple Flows of Control

The multiscalar paradigm is specially geared to execute multiple flows of control. While deciding the tasks, the multiscalar compiler will, as far as possible, attempt to generate tasks that are control-independent of each other, so that the multiscalar hardware can follow independent flows of control. However, most non-numeric programs have such hairy control structures that following multiple, *independent* flows of control is almost infeasible. So, our solution is to follow multiple, possibly control-dependent, and possibly data-dependent flows of control, in a *speculative* manner. Thus, the multiscalar compiler will, as far as possible, attempt to demarcate tasks at those points where the compiler may be fairly sure of the next task to be executed when control leaves a task (although it may not know the exact path that will be taken through each task at run time). Such a division into tasks will not only allow the overall large window to be accurate, but also facilitate the execution of (mostly) control-independent code in parallel, thereby allowing multiple flows of control, which is needed to exploit significant levels of ILP in non-numeric applications [90]. By encompassing complex control structures within a task, the overall prediction accuracy is significantly improved.

3.3.3. Speculative Execution

The multiscalar paradigm is an epitome for speculative execution; almost all of the execution in the multiscalar hardware is speculative in nature. At any time, the only task that is guaranteed to be executed non-speculatively is the sequentially earliest task that is being executed at that time. There are different kinds of speculative execution taking place across tasks in the multiscalar hardware: (i) speculative execution of control-dependent code across tasks, and (ii) speculative execution of loads before stores from preceding tasks, and stores before loads and stores from preceding tasks. The importance of speculative execution for exploiting parallelism in non-numeric codes was underscored in [90].

3.3.4. Parallel Execution of Data Dependent Tasks

Another important feature and big advantage of the multiscalar paradigm is that it does not require the parallelly executed tasks to be data independent either. If inter-task dependencies are present, either through registers or through memory locations, the hardware automatically enforces these dependencies. This feature gives significant flexibility to the compiler. It is worthwhile to point out, however, that although the execution of (data) dependent tasks can be overlapped, the compiler can and should as far as possible attempt to pack dependent instructions into a task, so that at run time the tasks can be executed mostly independent of each other. This will help to reduce runtime stalls, and improve the performance of the processor.

3.3.5. Exploitation of Localities of Communication

The decentralization of critical resources cannot be brought about by merely distributing the hardware; as far as possible, information should not be flowing all across the hardware. By grouping data dependent instructions into a task and by making tasks as independent as possible, much of the communication in the processor can be localized to within each task execution unit. This allows the multiscalar hardware to exploit the localities of communication present in a program.

3.3.6. Conveyance of Compile-Time Information to the Hardware

In traditional processing paradigms involving some form of data-driven execution, although the compiler has access to different types of information such as register dependencies and control flow in the program, these information are not directly conveyed to the hardware through the program. The hardware has to painstakingly reconstruct by instruction decoding the information that was available at compile time. It would be ideal if some of the information available at compile time is conveyed to the dynamic scheduling hardware. By grouping a set of instructions as a single unit (task), the multiscalar compiler conveys to the hardware additional information that allows the hardware to make more informed run-time decisions about the flow of program control than it would be possible if it were to determine the control flow through the decoding of instructions. Similarly, the compiler can convey information to the hardware by a bitmap the registers that are updated in a task. The hardware can use this bitmap in a simple manner to determine inter-task register dependencies (explained in chapter 5). Thus, the hardware need not reconstruct some of the mundane dependency information that is available at compile time.

3.4. Comparison with Other Processing Paradigms

3.4.1. Multiprocessing Paradigm

The multiscalar paradigm has some striking similarities to the conventional multiprocessing paradigm. It is important to see how the two paradigms differ from each other. In the case of a multiprocessor, the compiler determines the independence of tasks. At run time, each task is assigned to a different processor of the multiprocessor, effectively establishing a large dynamic window of several tasks. Each processor executes the assigned task sequentially; collectively the multiprocessor executes several instructions per cycle, one from each task. While no static code motions have been made in the window, multiprocessor execution moves some instructions ‘‘up in time’’. That is, instructions from later in the sequential instruction stream are initiated earlier in time; this motion is facilitated by guarantees from the compiler that the instructions are independent.

Thus in a multiprocessor, independence of tasks — both data independence and control independence — is guaranteed by the compiler. Therefore, the multiple tasks are not executed in a speculative manner. Whenever there is an inter-task data dependency, the compiler inserts explicit synchronization instructions. The multiscalar paradigm, on the other hand, does not require the independence of tasks — both data independence and control independence. It overcomes inter-task control dependencies by speculative execution, and enforces inter-task data dependencies with the help of fast low-level inter-task synchronization mechanisms. Table 3.1 succinctly shows the similarities and differences between the multiscalar and multiprocessing paradigms (both shared-memory

Table 3.1. Comparison of Multiprocessor and Multiscalar Paradigms

Attributes	Multiprocessor	Multiscalar
Task determination	Static	Static (Can be dynamic)
Static guarantee of inter-task control independence	Required	Not required
Static guarantee of inter-task data independence	Required	Not required
Inter-task synchronization	Explicit	Implicit
Medium for inter-task communication	Memory (if shared-memory) Messages (if message-passing)	Through registers and memory
Register space for PEs	Distinct	Common
Memory space for PEs	Common (if shared-memory) Distinct (if message-passing)	Common
Speculative execution	No	Yes
Multiple flows of control	Yes	Yes

and message-passing multiprocessing).

3.4.2. Superscalar Paradigm

The multiscalar paradigm has several similarities with the superscalar paradigm too. Both use control-driven specification, and a combination of data-driven execution and control-driven execution. Figure 3.7 highlights the fundamental differences between the two paradigms. A significant difference is that the superscalar views the entire dynamic window as a centralized structure, whereas the multiscalar views the dynamic window as distributed tasks. Therefore, the superscalar paradigm is unable to exploit communication localities within the dynamic window. This restricts the size of the window; lesser the window size, lesser the parallelism exploited. Moreover, the superscalar performs only data-driven execution, and no control-driven execution among the instructions in the window. The overhead of a full-blown data-driven execution also restricts the size of the dynamic

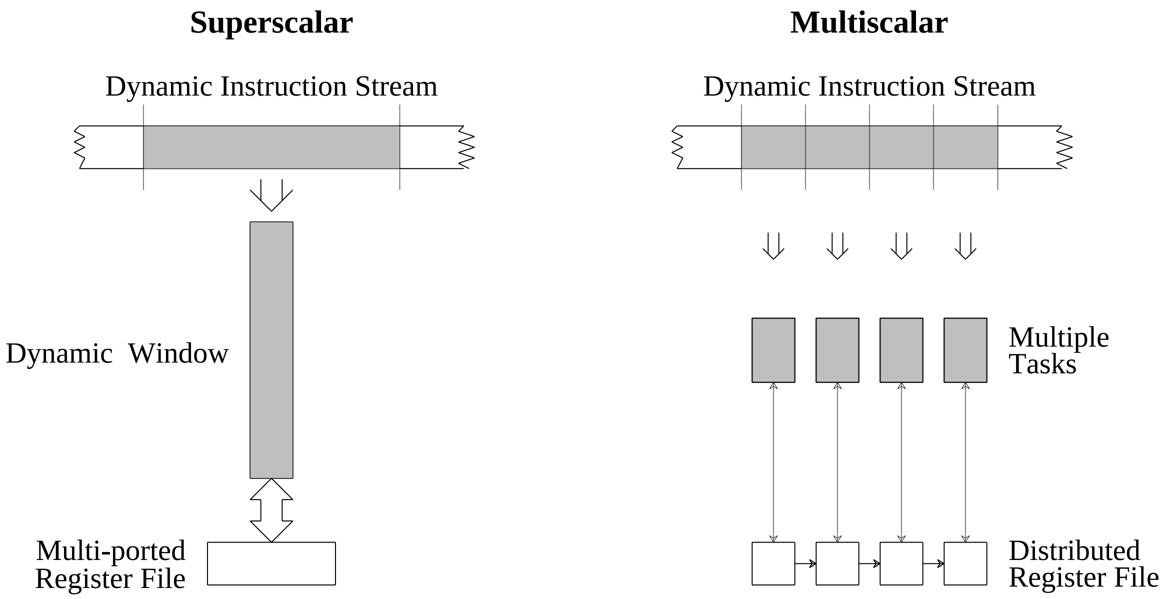


Figure 3.7: Comparison of Superscalar and Multiscalar Paradigms

window. Another major distinction between the superscalar and the multiscalar paradigms is that the superscalar pursues a single flow of control, whereas the multiscalar pursues multiple flows of control. Table 3.2 succinctly shows the similarities and differences between the superscalar and multiscalar paradigms.

Table 3.2. Comparison of Superscalar and Multiscalar Paradigms

Attributes	Superscalar	Multiscalar
Program specification order	Control-driven	Control-driven
Parallelism specification	Not specified	(Possibly dependent) tasks
Instruction fetching	Control-driven	Control-driven
Program execution order	Data-driven	Control-driven & Data-driven
Dynamic window	Centralized	Distributed
Exploiting localities of communication	No	Yes
Multiple flows of control	No	Yes
Static extraction of parallelism	Helpful	Helpful

3.4.3. VLIW Paradigm

It is also interesting to compare the multiscalar paradigm with the VLIW paradigm. In the VLIW paradigm, *independent* operations are horizontally grouped into instructions, whereas in the case of the multiscalar paradigm, *mostly dependent* operations are vertically grouped into tasks. This distinction is clearly illustrated in Figures 3.8(i) and (ii), which show 16 operations each for the VLIW and multiscalar paradigms; the same code is used for both paradigms. After the completion of each instruction, the paradigm guarantees that the results of all four operations in the instruction are available to all operations of the next instruction in the next clock cycle. Thus, the VLIW hardware needs to provide a crossbar-like inter-operation communication mechanism. Now consider the multiscalar execution. When an instruction is completed in a task, the result is made available to only the same task and possibly to the subsequent task in the next cycle. If a distant task needs the result, the results will be forwarded one task at a time. Table 3.3 succinctly shows the similarities and differences between the VLIW and multiscalar paradigms.

3.4.3.1. Adaptability to Run-Time Uncertainities

Let us see how the multiscalar paradigm and the VLIW paradigm compare in terms of adaptability to run-time uncertainities. In the case of the VLIW paradigm, the entire processor stalls when an unpredicted events such as data cache misses, instruction cache misses, and memory bank conflicts occur. The delay in the completion of any one of the operations in a horizontal instruction delays the completion of the entire instruction, because all operations are processed in lock-step. Thus, the

Table 3.3. Comparison of VLIW and Multiscalar Paradigms

Attributes	VLIW	Multiscalar
Program specification order	Control-driven	Control-driven
Parallelism specification	Horizontal Instructions	(Possibly dependent) tasks
Instruction fetching	Control-driven	Control-driven
Program execution order	Control-driven	Control-driven & Data-driven
Static extraction of parallelism	Critical	Helpful
Dynamic extraction of parallelism	No	Yes
Multiple flows of control	No	Yes
Adaptation to run-time Uncertainities	Poor	Good

VLIW performance can degrade significantly when run-time predictability reduces. Figures 3.8(iii) and (iv) show one possible execution of the operations of Figures 3.8(i) and (ii) in a 4-way VLIW machine and a 3-unit multiscalar machine. In these two figures, time increases in the downward direction. Notice that in Figure 3.8(iv), unit 1 executes task 4 after it finishes the execution of task 1. The 4 independent operations in each VLIW instruction are executed in lock-step. In Figure 3.8(iii), when the load in VLIW instruction 2 encounters a cache miss or a memory bank conflict, the entire VLIW processor stalls, whereas the same incident in the multiscalar paradigm causes only execution

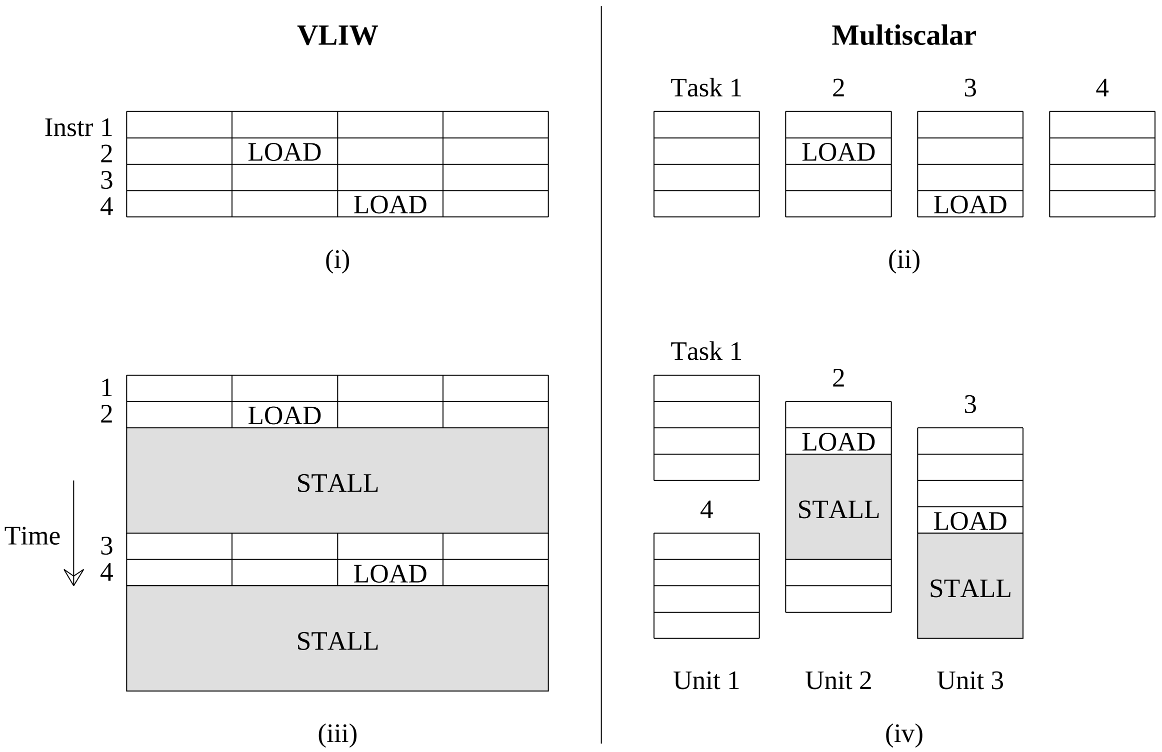


Figure 3.8: Stalls in VLIW and Multiscalar Paradigms due to Run-Time Uncertainities

unit 2 to stall. Thus, we can see that the multiscalar processor is better adapted to run-time uncertainities than the VLIW paradigm, because it does not stall the entire processor when untoward run-time incidents occur.

3.5. The Multiscalar Processor

In developing the conceptual basis for the multiscalar paradigm, we had purposely attempted to factor out implementation issues. Perhaps now is the time to start addressing the implementation issues, and throw more light on how the different aspects of the paradigm can be implemented. We like to emphasize that the *multiscalar processor* described in the ensuing chapters of this thesis is only one possible implementation of the multiscalar paradigm. Throughout the design of this implementation we have emphasized two points — *decentralization* (which facilitates expandability) and *realizability*. Several novel techniques are used to decentralize the resources, without which the potential of the multiscalar paradigm could not have been exploited. The techniques used for decentralizing different parts of the system are different, because the way these parts work and fit into the system are also different.

From the early stage of the design itself, we were very much concerned with the realizability of the hardware. The core part of the implementation is a circular queue of identical *execution units*, each of which is equivalent to a typical datapath found in modern processors. And what could be more convenient than replicating a sequential processor — something we know how to design well — and connecting several of them together as a circular queue? Figure 3.9 illustrates how multiple execution units can be connected together to form a multiscalar processor.

The fundamentals of the multiscalar processor's working are best understood as follows. The execution unit at the tail of the queue is assigned a task (explained in chapter 4); it executes the task, fetching, decoding, and executing the instructions in the task, one by one, just as a traditional sequential processor would. A global control unit determines (predicts) which task will be executed next, and assigns it to the next execution unit in the next cycle. The active execution units, the ones from

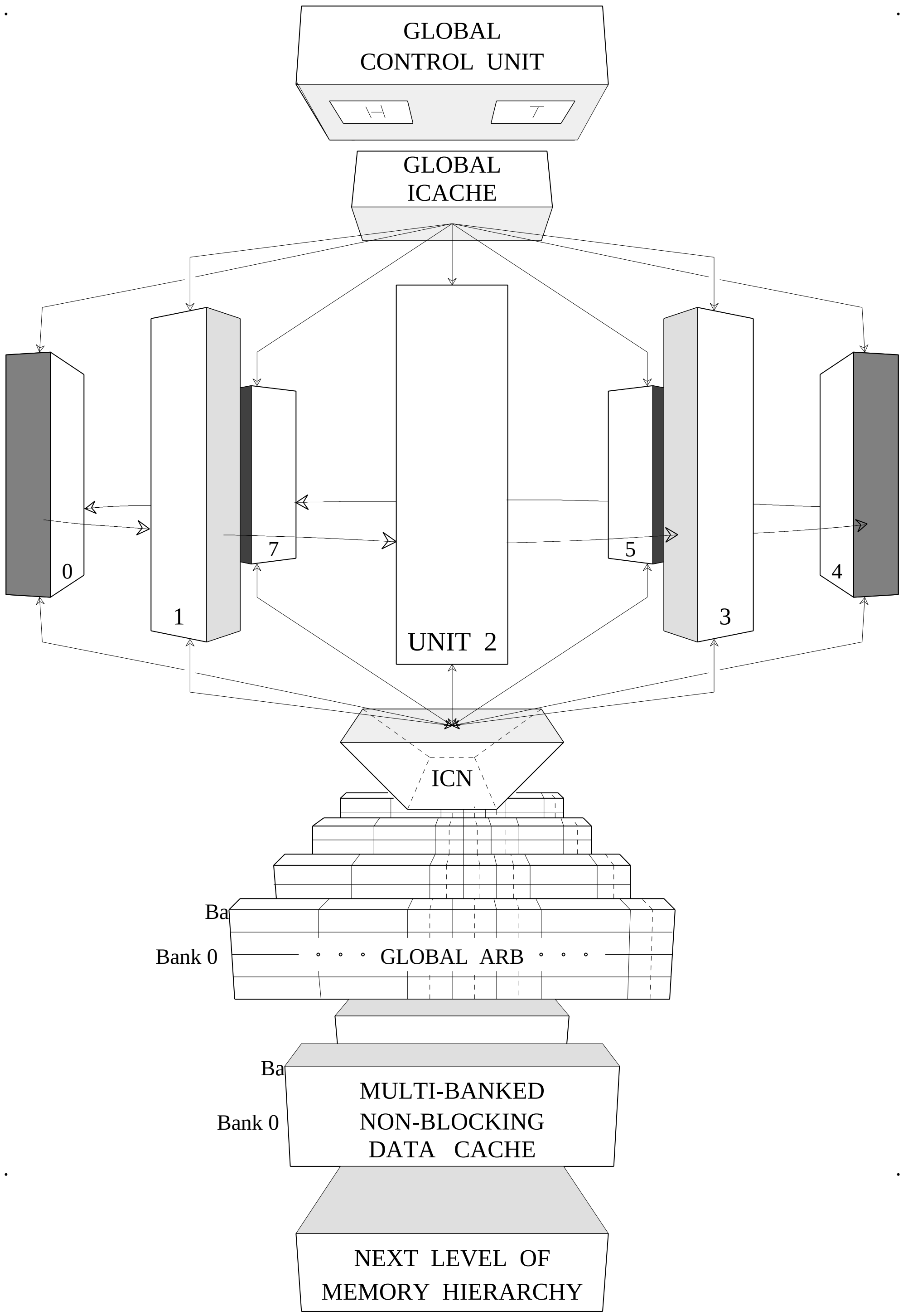


Figure 3.9: Block Diagram of an 8-Unit Multiscalar Processor

the head to the tail, together constitute the large dynamic window of instructions, and the units contain tasks, in the sequential order in which the tasks appear in the dynamic instruction stream. When control flows out of the task in the execution unit at the head, the head pointer is moved forward to the next execution unit.

The multiscalar processor consists of several building blocks, as shown in Figure 3.9. Almost all the major functions in the multiscalar processor are broken down into two parts — one for carrying out the intra-unit part of the function and the other for carrying out the inter-unit part. The intra-unit part is easily decentralized because it is distributed throughout the processor. The inter-unit part may or may not be decentralized depending on whether the communication it handles is localized or global in nature. For instance, the inter-unit register communication is easily decentralized (c.f. section 5.3) because much of the register communication in the multiscalar processor is localized [59].

The building blocks of the multiscalar processor are described in great detail in the following three chapters. Chapter 4 describes some of the major parts such as the execution units, the control mechanism, and the instruction supply mechanism. The inter-operation communication mechanism is described in chapter 5, and the memory system is described in chapter 6.

3.6. Summary

We have proposed a new processing paradigm for exploiting ILP, called the multiscalar paradigm. The basic idea of the paradigm is to consider a block of instructions as a single unit (task), and exploit ILP by executing many such tasks in parallel. The parallelly executed tasks need not be be independent, and can have both control dependencies and data dependencies between them. Each task can be as general as any connected subgraph of the control flow graph of the program being executed. The multiscalar paradigm is the fruit born from the fusion of static scheduling and dynamic scheduling; it uses control-driven specification, and a combination of data-driven and control-driven execution. In this regard, it shares a number of properties with the superscalar paradigm and the restricted dataflow paradigm. The essence of data-driven execution is captured by simple data forwarding schemes for both register and memory values. The fundamental properties of control-driven specification that we retained includes a sequential instruction stream, which relies on inter-instruction communication through a set of registers and memory locations. The result is a simple paradigm that accepts sequential code, but behaves as a fairly restricted dataflow machine. The splitting of the dynamic window into tasks allows the decentralization of the hardware resources, and the exploitation of localities of communication. For this reason, we believe that the multiscalar processor is conceptually completely different from the previous ILP processors of whatever generation.

We have also started describing a processor implementation of the multiscalar paradigm. In our view, the beauty of the multiscalar processor lies in its realizability, not to mention its novelty. Essentially, we have taken several sequential processors, connected them together, and provided some additional hardware support mechanisms. As we will see in the next three chapters, it draws heavily on the recent developments in microprocessor technology, yet goes far beyond the centralized window-based superscalar processors in exploiting irregular ILP. It has no centralized resource bottlenecks that we are aware of. This is very important, because many existing execution models are plagued by the need for centralized resources. Almost all the parts of our implementation are found in conventional serial processors, the only exception is the Address Resolution Buffer (ARB) described in chapter 6; yet these parts have been arranged in such a way as to extract and exploit much more parallelism than was thought to be realistically possible before. Another feature of the multiscalar processor is its expandability. When advances in technology allow more transistors to be put on a chip, the multiscalar implementation can be easily expanded by adding more execution units; there is no need to redesign the implementation/architecture.

Chapter 4. Control and Execution Units

How to connect multiple processors together in a tightly synchronized manner?

The previous chapter introduced the multiscalar processor and began discussing its overall working. The central idea, as we saw, is to connect several execution units as a circular queue, and assign a task to a separate execution unit. This chapter discusses some of the major parts of the multiscalar processor such as the execution units, the control mechanism, and the instruction supply mechanism.

Chapter 4 is organized into four sections as follows. Section 4.1 describes the execution unit. Section 4.2 describes the distributed control mechanism. Section 4.3 describes the distributed instruction supply mechanism, and section 4.4 presents the summary.

4.1. Execution Unit

The multiscalar processor consists of several identical execution units, as described in chapter 3. Each execution unit is responsible for carrying out the execution of the task assigned to it. Figure 4.1 gives the block diagram of an execution unit. It consists of a control unit (local CU), an instruction cache (local icache), an issue and execution (IE) unit, a register file (local RF), a forwarding queue (FQ), and a memory disambiguation unit (local ARB). The local CU is used for sequencing the control through a task. and is similar to the control unit of an ordinary processor. The IE unit is described in this section. The RF is used to carry out the inter-operation communication, and is described in section 5.3. The FQ is used to forward register values produced in an execution unit to the subsequent execution units, and is described in section 5.3. The local ARB is used to do memory disambiguation within an unit, and is described in section 6.4.5. Notice the remarkable similarity

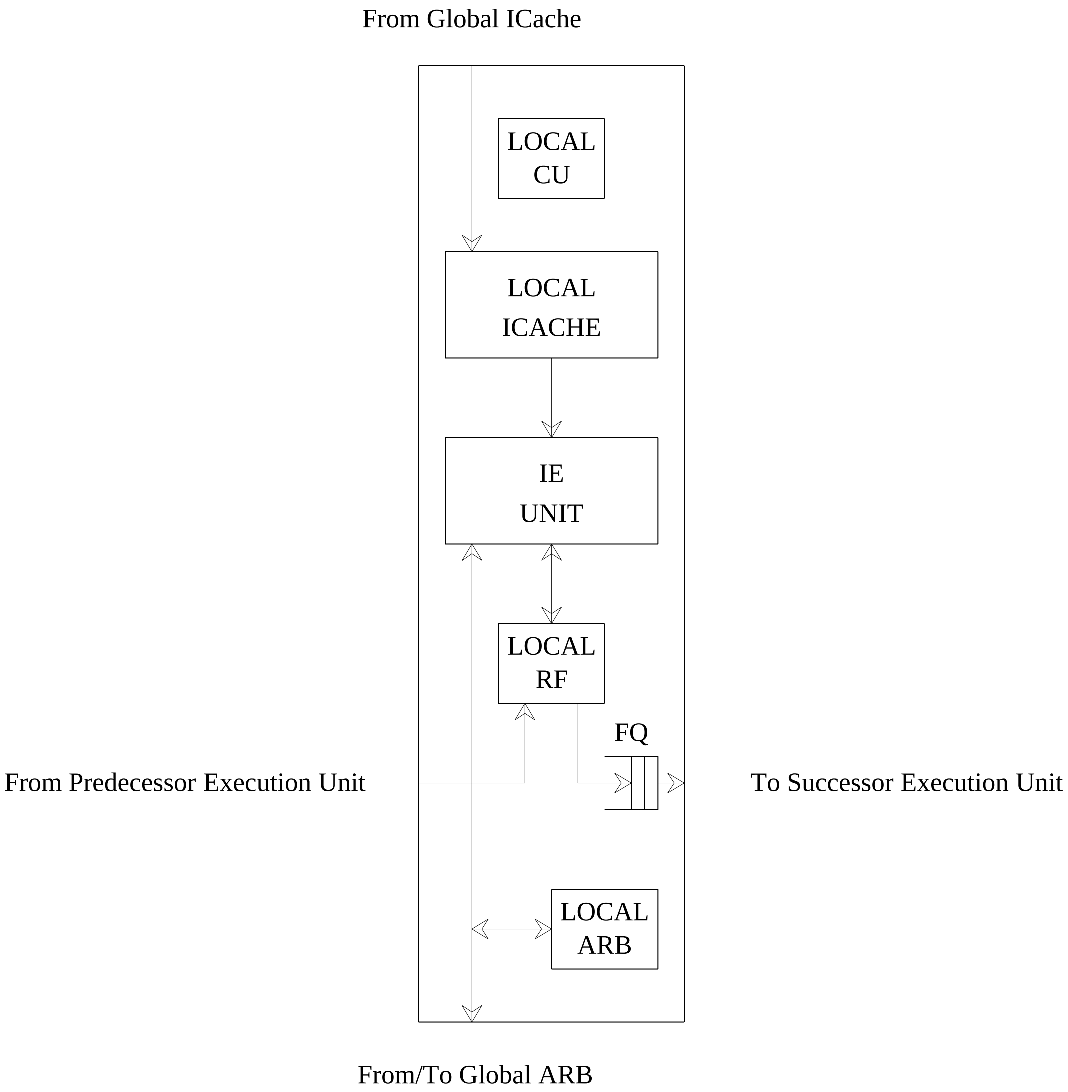


Figure 4.1: Block Diagram of an Execution Unit

between an execution unit and an ordinary, execution datapath; the only singular feature in a multiscalar execution unit is the forwarding queue connecting an execution unit to the successor execution unit.

IE Unit: Each execution unit has as its heart an Issue and Execution (IE) unit, which fetches instructions from the local instruction cache, and pumps them to its functional units after resolving their data dependencies. The IE unit is comparable to the Instruction Issue and Execution Unit of a conventional processor, and has its own set of functional units. In any given cycle, up to a fixed number of ready-to-execute instructions begin execution in the IE units of each of the active units. It is possible to have an arbitrary execution model within an IE unit; the type of execution model used within an IE unit depends on the type of parallelism one expects to exploit within a task, given that the compiler can attempt to pack dependent instructions within a task (c.f. chapter 9).

4.2. Control Mechanism

The control mechanism is divided into two parts: intra-task control and inter-task control. The intra-task control is carried out by local control units in the execution units, and the inter-task control is carried out by a global control unit. Figure 4.2 illustrates this. The global control unit manages the head and tail pointers of the circular queue of execution units. It also performs the job of assigning tasks and deassigning tasks to the units. After assigning a task to the tail execution unit, the global control unit predicts (based on static hints or dynamic history) the next task to be executed. Every cycle, a new task is assigned to the execution unit at the tail, and the tail pointer advanced by one execution unit, unless the circular unit queue is full. When all the instructions in the execution unit at the head have completed execution, the control unit deassigns (commits) that task, and moves forward the head pointer to the next execution unit.

It is important to note that all that the global control unit does when it assigns a task to an execution unit is to tell the unit to execute the task starting at a particular PC (program counter) value; it is up to the execution unit to fetch the required instructions, decode them and execute them (most likely in serial order) until control flows out of the task (c.f. section 4.2.3). The global control unit does not perform instruction decoding. (A major purpose of ‘‘decoding’’ instructions in a dynamically scheduled ILP processor is to establish register dependencies between instructions. We shall see in section 5.3.2 how the multiscalar processor enforces the register dependencies without decoding the instructions.) Because the global control unit’s chore is relatively straightforward, it does not

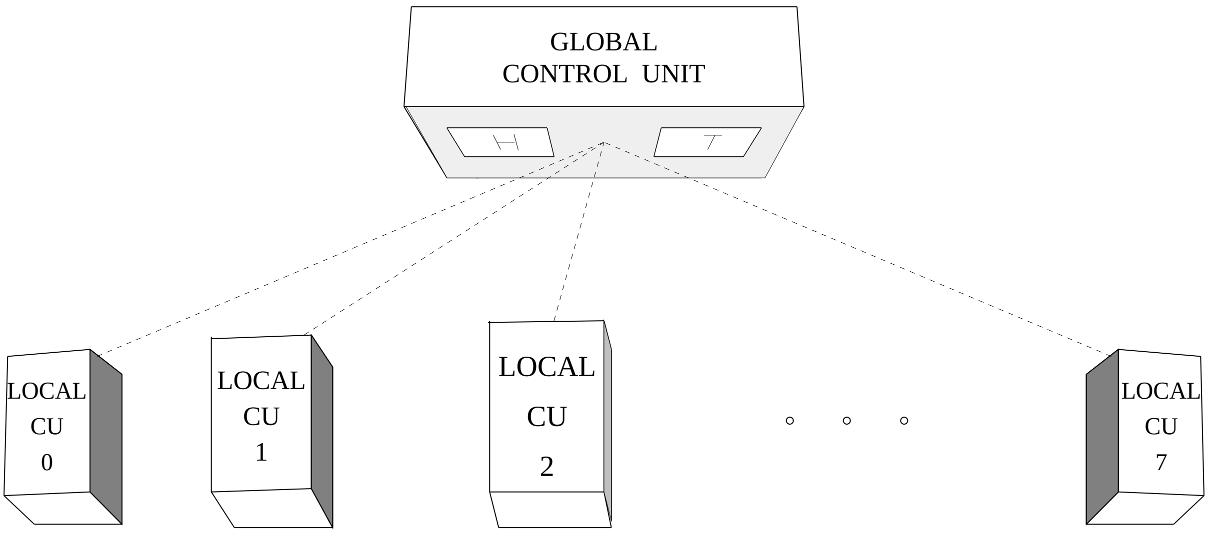


Figure 4.2: Distributed Control Mechanism for an 8-Unit Multiscalar Processor

become a potential bottleneck. Control units with instruction decoders that feed centralized windows are a major impediment to performance in superscalar processors, as shown in [140].

4.2.1. Task Prediction

After a task is assigned to an execution unit, the global control unit needs to predict the next task to be executed. A task may have multiple targets, and the global control unit has to make an intelligent prediction (based on static hints or run-time history), and choose one target. A highly accurate prediction mechanism is very important, as the construction of a large window of tasks requires many task predictions in a row; by the time the predictor reaches execution unit *n*, the chance of that unit getting assigned the correct task is only p^n, where *p* is the average task prediction accuracy. Thus even an average task prediction accuracy of 90% is a far cry from what is required. Achieving a reasonably good *task prediction* accuracy may not be as difficult as achieving a good *branch prediction* accuracy, if many of the unpredictable branches in the program have been encapsulated within the multiscalar tasks.

4.2.1.1. Static Task Prediction

There are several ways of doing task prediction. One option is for the compiler to do a static prediction and convey it to the hardware along with the task specification. With this scheme, it is possible to let the local control units perform all the functions carried out by the global control unit, including the assigning of tasks to the execution units. That is, each local control unit can, depending on the task assigned to its execution unit, make a prediction about the task to be executed on the subsequent execution unit, and inform that unit. The managing of the head and tail pointers can also be done by the local control units. The advantage of this scheme is that the global control unit can be completely distributed among the local control units to obtain a truly distributed control processor. This distribution would be difficult with dynamic prediction (discussed next), as the dynamic task predictor needs to have a global picture regarding the tasks allocated to all execution units to have a reasonable prediction accuracy. The feasibility of static prediction (in terms of the obtainable task prediction accuracy) needs further study.

4.2.1.2. Dynamic Task Prediction

Dynamic task prediction can use techniques similar to those used for dynamic branch prediction. In particular, it is possible to keep a condensed history of previous task outcomes, and base decisions on the history. The type of condensation used can vary — for instance, a set of counters to keep track the number of times each target of a task was previously taken, or a two-level scheme with a register keeping track of the last several outcomes, and a set of counters keeping track of the number of times each target was taken for different patterns of previous outcomes [166]. For branch predictions, the later scheme has been shown to be very effective, giving branch prediction accuracies to the tune of 95% for non-numeric programs.

Another issue to be addressed in dynamic task prediction is the issue of basing prediction decisions on *obsolete history*. Irrespective of the type of history information stored, the control unit will often need to make several task predictions in a row, without being able to update the task history table in between, because the intervening outcomes may have not been determined. Because of basing the prediction on obsolete history information, the prediction accuracy might decrease when several predictions are made in a row. This problem can be overcome by speculatively updating the history information as and when a task prediction is made. If a prediction is later found to be incorrect, the speculative updates to the history tables can be undone. If the average prediction accuracies are very high, the speculative updates are almost always correct. If using a more recent history is better than using an obsolete history to make prediction decisions, then using speculatively updated information is better than using obsolete information to make prediction decisions in the multiscalar processor. Let us see why this is indeed the case.

Let the task history at some stage of execution be H_1. Let a prediction P_1 be made to execute a task T_1 in execution unit E_1. The new history obtained by speculatively updating H_1 with P_1 is given by

$$H_2 = f(H_1, P_1)$$

where f is the history update function. Assume that the next prediction, P_2, is made on H_2 rather than on H_1. Let us analyze the effect of this decision. If P_1 is correct, then H_2 is not only correct but also more recent than H_1, and therefore P_2 is more likely to be correct than if it were based on H_1. If, on the other hand, P_1 is incorrect, then the multiscalar processor discards both T_1 and T_2 any way (c.f. section 4.2.2). Thus, basing the decision of P_2 on H_2 does not do any harm, provided a more recent history does not give a poorer prediction accuracy than an obsolete history.

Return Addresses: When a task contains return instruction(s), its targets may not be the same in different executions of the task. Although a history-based prediction mechanism predicts the targets fairly accurately for tasks whose boundaries are defined by branches, this mechanism cannot be used for predicting the target of procedure returns. For effectively predicting the return addresses, the global control unit uses a stack-like mechanism similar to the one discussed in [81]. When the predictor predicts a procedure call as the next task to be executed, the return address is pushed onto the stack-like mechanism. When the predictor predicts a procedure return as the next task to be executed, the return address is popped from the stack. The stack mechanism is updated as and when actual outcomes are known.

4.2.2. Recovery Due to Incorrect Task Prediction

When the task prediction done for allocating a task to execution unit *i* is found to be incorrect, recovery actions are initiated. Recovery involves discarding all the computations being carried out in the active execution units including and beyond *i*. The control unit moves the tail pointer to point to execution unit *i*, which had the incorrect task assigned. The speculative state information in the execution units stepped over by the tail pointer are also discarded. Chapters 5 and 6 describe how exactly the speculative register values and speculative memory values are stored in the multiscalar processor, and discarded at times of recovery. We will see the advantage of storing state information at task boundaries, which facilitate fast recovery actions.

Notice that when a task prediction is found to be incorrect, the work going on in all subsequent active execution units is discarded, and tasks are again allocated starting from the execution unit which was executing the incorrect task. There is a possibility that some of the discarded tasks may be reassigned to the same units later; however, determining this information and to do appropriate recovery is difficult. The decision to discard all subsequent tasks was made in favor of simple and fast recovery.

4.2.3. Task Completion

The local control unit determines when the execution of the task in its execution unit is complete, and informs the global control unit of this event. Because a task could be an arbitrary connected subgraph of a program's CFG, it can have complex control flows within it, and there may be exits out of the task from the middle of the task. The execution of a task can be said to be complete when control flows out of the task. Detection of this condition is intimately tied to the manner in which tasks are specified by the compiler to the hardware. For instance, if the compiler specifies a task by providing the task's entry point and list of targets, then each time a branch or a jump instruction is executed, the local control unit can check if control flows to one of the targets in which case the execution of the task is over. Execution of a function call instruction or a return instruction can also indicate the completion of the task, if a task does not extend over multiple functions.

When the global control unit is notified that the execution unit at the head has completed the execution of the task assigned to it, the global control unit advances the head pointer by one execution unit. Sections 5.2.3.4 and 6.4.4 describe further actions taken by the register file mechanism and the data memory mechanism to commit speculative values to the architectural state of the machine.

4.2.4. Handling Interrupts

When a program is being executed, two types of external interrupts can occur: higher priority interrupts and lower priority interrupts. Depending on the priority of the interrupt, the action taken is also different. If the interrupt is of higher priority than that of the current program being executed, then all the active tasks are relinquished (using the recovery action described above), and control is immediately handed over to the interrupt handler. The first task from the interrupt handler then runs at the head unit, the second task at the successor unit, and so on. When the interrupt servicing is over, control again returns to the original program. The task that was being executed at the head when the interrupt arrived is reassigned and re-executed from its beginning.

If the interrupt is of lower priority, the active tasks are not relinquished; only the control is passed to the interrupt handler. The first task from the interrupt handler then runs at the tail unit (when it becomes free), the second task at the successor unit, and so on, in parallel to the already executing tasks of the original program.

4.2.5. Handling Page Faults

Special care is needed in handling a page fault, as the page fault is an *internal* event, having a bearing to a particular PC value within a multiscalar task. This has implications while designing a protocol for handling page faults in the multiscalar processor, because the processor has to treat it as a precise interrupt. If not handled properly, livelock can occur. Consider that a page fault occurred in execution unit *i*. If the protocol is to squash all units from *i* onwards and up to the tail, then livelock will occur if there are *m* memory accesses in unit *i*, all of them are to different pages, and the main memory has < *m* pages. Generally, main memory will have far more pages than the number of pages accessed by the memory references in a task, and the occurrence of a livelock may only be a theoretical possibility. Livelocks are more likely to occur if all units are squashed before running the page handler.

In order to avoid livelock, the protocol *should* squash all the instructions and tasks beyond the faulting memory reference, and *no other* instructions and tasks. After evicting these instructions and tasks, the page handler can be run in unit *i*'s successor unit. Once the page handler has completed, it should re-execute the faulting memory reference and complete the execution of the task that contains the faulting reference.

4.3. Instruction Supply Mechanism

The execution of multiple tasks in parallel will bear fruit only if instructions are supplied to the execution units at an adequate rate. Supplying multiple execution units in parallel with instructions from different tasks can be difficult with an ordinary centralized instruction cache. The multiscalar processor therefore uses a two-level instruction cache. The local instruction caches in the execution units correspond to the first level (the level closest to the execution units) of the cache hierarchy. Thus, the local instruction caches are distributed throughout the processor, (as shown in Figure 4.3 for an *n*-unit multiscalar processor). The local instruction caches are backed up by a global instruction cache.

During the execution of a task, the IE unit of the execution unit accesses instructions from its local instruction cache. If a request misses in the local cache, the request is forwarded to the global instruction cache. If the request misses in the global cache, it is forwarded to main memory. If the task is available in the global cache, it is supplied to the requesting local cache. A fixed number of

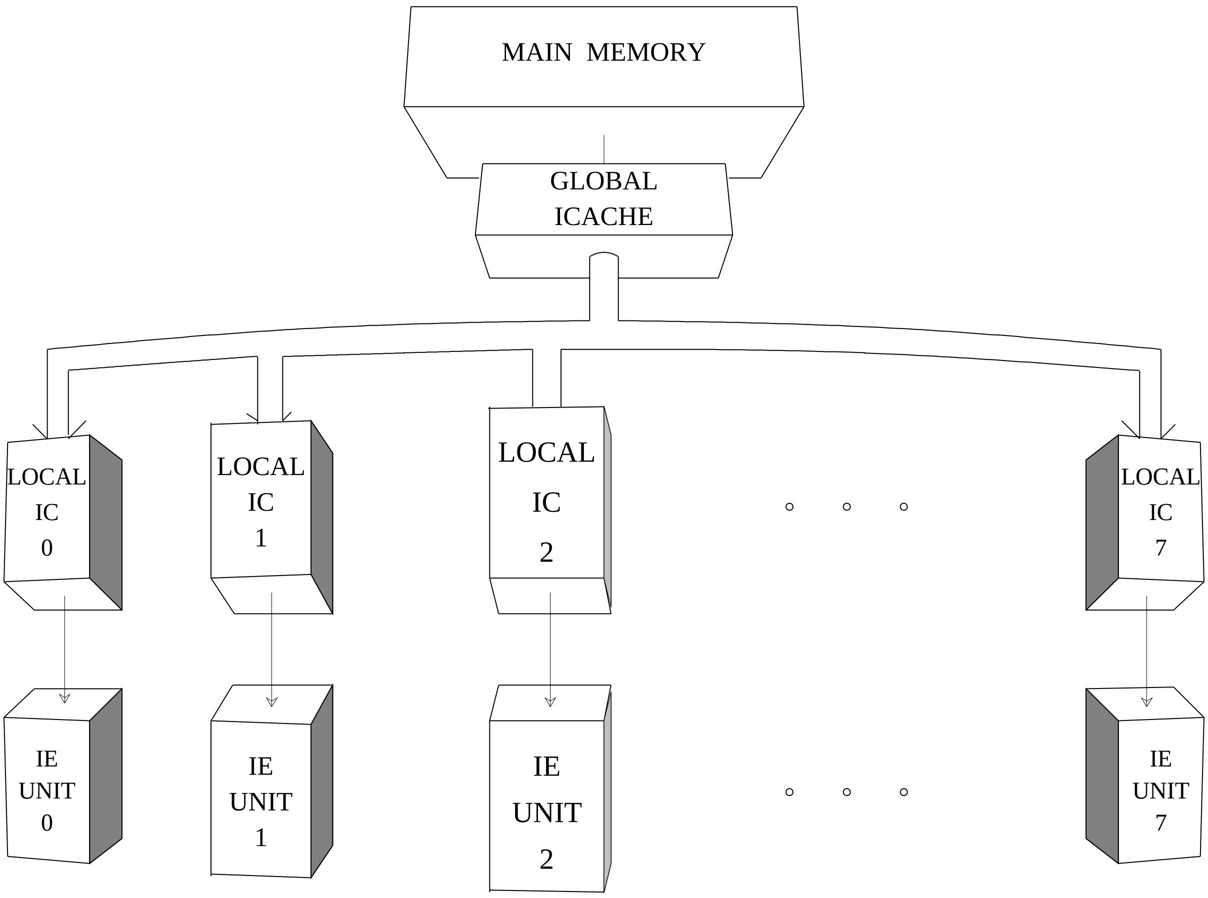


Figure 4.3: Distributed Instruction Supply Mechanism for an 8-Unit Multiscalar Processor

instructions are transferred per cycle until the entire task is transferred — a form of intelligent instruction prefetch. If the transferred task is the body of a loop, the local caches of the subsequent units can also grab a copy of the task in parallel, much like the *snarfing* (*read broadcast*) scheme proposed for multiprocessor caches [64].

Notice that several IE units can simultaneously be fetching instructions from their corresponding local instruction caches, and several local caches can simultaneously be receiving a task (if the task is the body of a loop) from the global cache, in any given cycle. Further, if a miss occurs in any of the instruction caches, the execution units that are already in execution need not stall, and can continue with their execution. This illustrates the decentralization concept that we shall see throughout this thesis.

4.4. Summary

This chapter described the execution units, the control mechanism, and the instruction supply mechanism of the multiscalar processor. The execution units are identical, and each execution unit is remarkably similar to an ordinary, datapath. An execution unit contains a control unit, an instruction cache, an issue and execution (IE) unit, a register file, and a memory disambiguation unit. The main block in the execution unit is the IE unit, which performs the fetching, decoding, and execution of the instructions in the task assigned to the execution unit.

The control mechanism consists of a global control unit and several local control units, one per execution unit. Although the global control unit can be termed the multiscalar processor's brain, much of the control functions related to task execution are carried out on a local basis by the local control units, thereby decentralizing most of the control decisions in the processor. For instance, all that the global control unit does when it assigns a task to an execution unit is to tell the unit to start the execution of the task beginning at a particular PC. It is up to the execution unit to fetch and execute the instructions of the task until control flows out of the task, and then convey this information to the control unit.

The instruction supply mechanism is also decentralized. Each execution unit has a local instruction cache from which its IE unit can fetch instructions. The local instruction caches are backed up by a common global instruction cache.

Chapter 5. Register File

How to exploit communication localities to decentralize the register file?

This chapter describes a decentralized register file for carrying out the inter-instruction communication in the multiscalar processor. The central idea of the decentralized register file scheme is to have multiple versions of the architectural register file so as to provide a hardware platform to store the speculative register values produced in the execution units (recall that much of the execution in the multiscalar processor is speculative in nature). The multiple versions also allow precise state to be maintained at inter-unit boundaries, which facilitates easy recovery actions in times of incorrect task prediction. Having multiple versions also has the benevolent side-effect that the number of read ports and write ports required of each register file is much less than that required with a single centralized register file.

The outline of this chapter is as follows. Section 5.1 describes the requirements of the multiscalar inter-instruction communication mechanism, and motivates the need to decentralize it. Section 5.2 presents the basic idea of the multi-version register file, and section 5.3 presents its detailed working. Section 5.4 describes the novel features of the multi-version register file, and section 5.5 summarizes the chapter.

5.1. Inter-Instruction Communication in the Multiscalar Processor

Most programs following a control-driven specification specify inter-instruction communication by means of an architectural register file. Each time a datum is written into a register, a new *register instance* is created. Succeeding reads to the register use the *latest register instance*, as given by the sequential semantics of the program. Thus, when an instruction *I* performs a read access for register

R, the value obtained should be the latest instance of *R* produced before *I* in a sequential execution of the program. If the latest instance has not been produced when the access is made (this can happen in pipelined processors and dynamically scheduled processors), then naturally instruction *I* waits. In the multiscalar processor, the latest instance of *R* may be produced either in the same task to which *I* belongs or in a preceding task whose execution is carried out in a different execution unit. In the former case, the register read access by instruction *I* has to obtain the latest instance of *R* produced in the same unit (*intra-unit communication*), and in the latter case, the read access has to obtain the *last possible instance* of *R* produced in a different unit (*inter-unit communication*). Thus, there are conceptually two types of inter-instruction communication occurring in the multiscalar processor. The former relates to the communication between a producer operation in a dynamic task and consumer operations in the same dynamic task. The latter relates to the communication between a producer operation in one dynamic task and consumer operations in logically succeeding dynamic tasks.

The requirements of the multiscalar inter-instruction communication mechanism can be grouped under two attributes: (i) high bandwidth and (ii) support for speculative execution. These two attributes are described in detail below.

5.1.1. High Bandwidth

The first issue that the inter-instruction communication mechanism of the multiscalar processor should address is the bandwidth that it should provide. Because a number of execution units, each containing many instructions, are simultaneously active, the inter-instruction communication mechanism must provide a high bandwidth. Although techniques such as split register files (separate integer and floating point register files) can provide some additional bandwidth, the inter-instruction communication bandwidth requirement of the multiscalar processor is much higher. Similarly, techniques such as multiple architectural register files, proposed for the TRACE /300 VLIW processor [34] are inappropriate for the multiscalar processor, which has the notion of a single architectural register file. Therefore, we need to provide a decentralized realization of a single architectural register file. The main criteria to be considered in coming up with a good decentralized scheme is that it should tie well with the distributed, speculative execution nature of the multiscalar model.

5.1.2. Support for Speculative Execution

Much of the execution in the multiscalar execution units is speculative in nature; at any given time, only the task executed in the head unit is guaranteed to commit. To carry out speculative execution, the results of speculatively executed operations must not update the non-speculative state of the machine. Thus, there should be a hardware platform for storing the speculative register values generated in the execution units. Going from the head unit towards the tail unit, there is an extra level of speculation being carried out at each unit, necessitating a separate physical storage to store the speculative state of each unit. Although techniques such as *shadow register files* [141, 142] provide a separate shadow register file per unresolved branch to support multiple levels of speculative execution in control-driven execution processors, they require a complex interconnect between the functional units and the register files, because a register read operation must have access to all physical register files. This runs counter to the multiscalar philosophy of decentralization of critical resources.

5.2. Multi-Version Register File—Basic Idea

Having discussed the important requirements of the multiscalar register communication mechanism, let us now turn our attention to meeting these requirements. To decentralize the register file, and to support multiple levels of speculation, we use separate hardware to carry out intra-task and inter-task communication (true to the multiscalar paradigm's general methodology of decentralization of critical resources).

The hardware structure that we use for carrying out register communication in the multiscalar processor is called a multi-version register file. The basic idea behind the multi-version register file is to provide each execution unit with a different version of the architectural register file. Thus, each execution unit has a separate *local register file*, as shown in Figure 5.1. The local register files serve as a hardware platform for storing the speculative register values produced in the execution units, and are the working files used by the functional units in the IE units. Let us see how the multi-version register file handles intra-task communication and inter-task communication.

5.2.1. Intra-Task Register Communication

All intra-task register communication occurring in an execution unit is carried out by means of a *current file* present in the unit's local register file. Thus, each execution unit has its own current file. When an instruction produces a new register instance, the value is written to the current file in its unit. Subsequent reads from the same task to that register take the latest instance from the current file. When a task is being executed in a unit, the current file stores the latest register instances produced during the execution of the task, and is similar in that respect to the architectural register file in a conventional processor. When no task is being executed in a unit, its current file has no valid

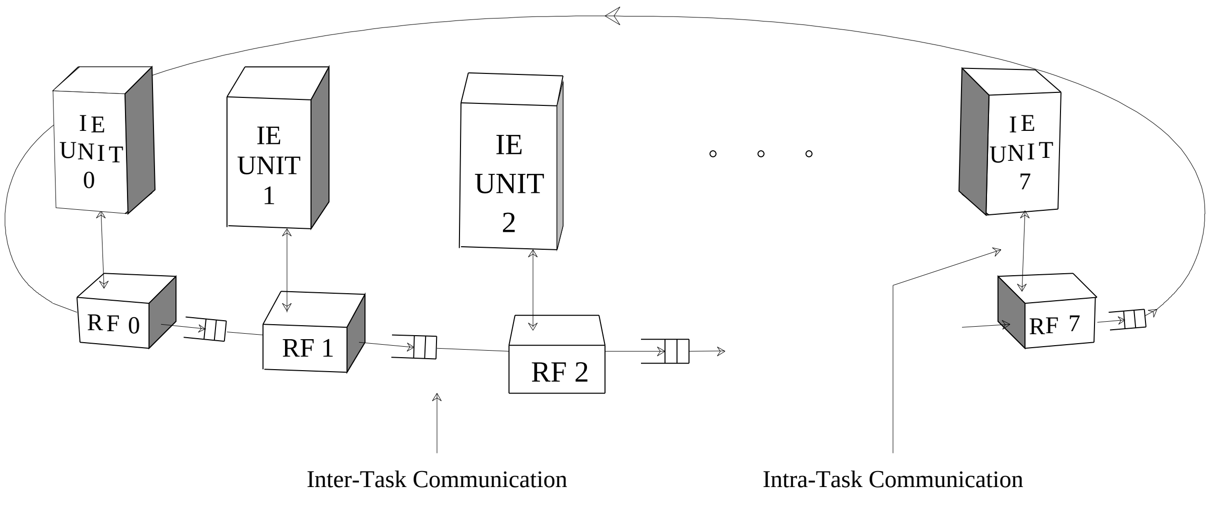


Figure 5.1: Multi-Version Register File for an 8-Unit Multiscalar Processor

entries. Because each unit has a separate current file, the intra-task register communication occurring in different units can all be carried out in parallel.

5.2.2. Inter-Task Register Communication

Next, let us see how inter-task register communication is carried out in the multiscalar processor. In a distributed environment, where multiple versions of a datum can be present (in multiple register files), there are 3 options for carrying out global communication. They are: (i) distributed writes and localized reads, (ii) localized writes and distributed reads, and (iii) distributed writes and distributed reads. In the first option, write operations update all the relevant non-local register files, and read operations proceed only to the local register file. In the second option, write operations update only the local register file, and read operations pick the correct values from among the non-local register files. The third option is a combination of the first two, in that both the writes and reads are performed over multiple register files. The second option has been used in the context register matrix scheme of Cydra 5 [127] and in the shadow register files of Torch [141, 142].

In the case of inter-task register communication, the first option has the advantage (over the other two) that an instruction requiring a register value need monitor only the local register file to determine if the latest instance is available, whereas in the other two options, the instruction has to keep monitoring the non-local register files. Therefore, we use the first option in the multiscalar processor. Our philosophy is that a register read operation should not proceed to the other units in search of the (latest) instance it needs, but that the latest instance should be made available to all the units. The way this is achieved is by forwarding from a task (to the subsequent tasks) the *last possible instances* of registers produced in that task, as and when the instances are determined to be the last possible instances. As per the sequential semantics of the program, only the last possible instances in a task will be required by subsequent tasks, if at all required.

The forwarding of register values is carried out with the help of a *unidirectional ring-type forwarding network*, which forwards values from one execution unit to the successor unit only, as shown in Figure 5.1. The advantage of this type of forwarding is that it does not require an expensive crossbar-like connection from every execution unit to the local register file of every other unit (as with the context register matrix scheme of Cydra 5).

When the latest register instances from the previous tasks arrive at an execution unit, instead of storing these values in the current file, they are stored in a separate file called the *previous file*. Thus the previous file is the hardware platform for storing the latest instances of the registers produced in the preceding tasks. The only reason for using separate register files for intra-task communication and inter-task communication is to ease recovery actions at times of incorrect task predictions.

Next we shall consider the last possible instances produced by subsequent tasks (in the subsequent units) of the active window. If these values are allowed to propagate beyond the tail unit and then onto the head unit and into the active window, they will defile the previous files of the active execution units with incorrect values. This not only will result in incorrect execution, but also will make it impossible to recover in times of incorrect task prediction. If, on the other hand, the values are stopped before the head unit, then each time a finished task is deallocated (committed), register values have to be copied to the head unit's previous file from the previous unit. The rate with which the register values are copied will decide the rate with which new tasks can be initiated. To avoid the significant overhead of continually copying register instances across (adjacent) execution units, a separate file, called the *subsequent file*, is used as a hardware platform for storing the register instances produced in the succeeding units of the active window. Thus, the subsequent file of an active execution unit maintains the register instances that are potentially required for the execution of the task allocated to the unit in the next round.

It is important to note that it is not a necessity to have three register files per execution unit. In particular, we saw that the subsequent file is primarily for performance. Similarly, the previous file can be merged with the current file of the same unit. However, recovery actions (in times of incorrect task prediction) become more complicated. Thus, the previous file is for fast recovery (*i.e.*, going back into the past), whereas the subsequent file is for fast progress (*i.e.*, going forward into the future).

5.2.3. Example

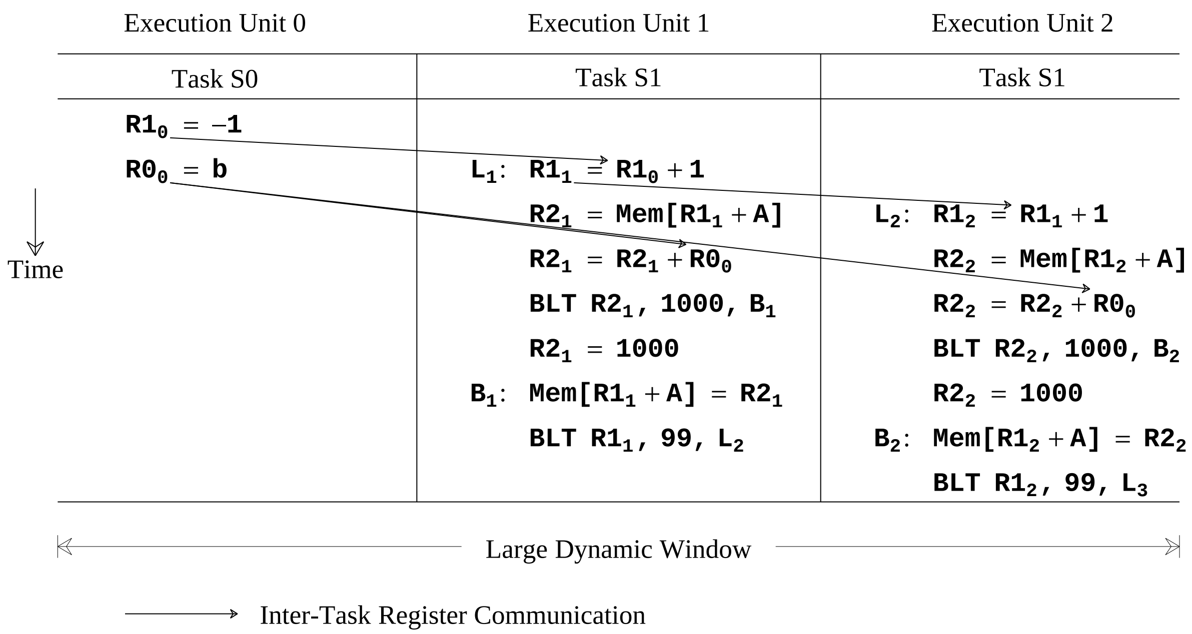
The basic idea of the multi-version register file is best illustrated by a simple example. Let us use the same assembly code of Example 3.1, which is reproduced in Figure 5.2(i). Register R0 is allocated for the loop-invariant value **`b`**, R1 is allocated for the loop induction variable **`i`**, and R2 is allocated for **`A[i]`**. Assume that the compiler has demarcated the pre-loop portion into a task (S0) and the loop body into another task (S1); at run time, multiple iterations of the loop become multiple dynamic tasks, as shown in Figure 5.2(ii). In the figure, the register instances produced by different tasks are shown with different subscripts, for example, $R1_0$, $R1_1$, and $R1_2$. For each dynamic task, R1 is defined in the previous task and R2 is defined within the task. R0 is an invariant for all dynamic tasks derived from the loop.

We shall see how different register instances are created and accessed when the tasks are executed. For ease of understanding, we will assume that serial execution is performed within each execution unit. First, consider the execution of task S0, in execution unit 0 (c.f. first column in Figure 5.2(ii)). The first instruction creates a new instance for register R0, namely $R0_0$, and this instance is written to the current file (present in the local register file) of unit 0. Because this instance is register R0's last possible instance produced in that task, the value is also forwarded to unit 1. When $R0_0$ arrives at unit 1 in the next cycle, the value is written to the previous file (present in the local register file) of unit 1. Because unit 1 does not produce any new instance for R0, it forwards $R0_0$ to unit 2. Thus, all execution units get the latest instance of R0, namely $R0_0$, and store it in their respective local register files.

Next, consider the execution of the second instruction in task S0, which creates a new instance for register R1, namely $R1_0$. The new instance is written into the current file of unit 0 as before. It is also forwarded to unit 1. When $R1_0$ arrives at unit 1 in the next cycle, it is stored in the previous file as before, but not forwarded to unit 2, as a new instance of R1 will be created in unit 1.

```
    R1  =  -1           ; initialize induction variable i
    R0  =  b            ; assign loop-invariant b to R0
L:  R1  =  R1 + 1       ; increment i
    R2  =  Mem[R1 + A]  ; load A[i]
    R3  =  R2 + R0      ; add b
    BLT  R3, 1000, B    ; branch to B if A[i] < 1000
    R3  =  1000         ; set A[i] to 1000
B:  Mem[R1 + A]  =  R3  ; store A[i]
    BLT  R1, 99, L      ; branch to L if i < 99
```

(i) Assembly Code



(ii) Multiscalar Execution

Figure 5.2: Example Code Illustrating the Basic Idea of Multi-Version Register File

Now, let us see what happens in parallel in unit 1, which is executing the second task (c.f. middle column of Figure 5.2(ii)). The execution of the first instruction generates a read access[3] for register R1. The correct instance needed in this case is $R1_0$, which is generated by the previous task, in execution unit 0. It is possible that $R1_0$ may not yet have arrived in unit 1, in which case a busy bit associated with R1 in unit 1's previous file will indicate that the correct instance has not yet arrived. If $R1_0$ is available in the previous file, then the busy bit will be reset, and the correct value can be read from the previous file. When this instruction produces a new instance of R1, namely $R1_1$, the new instance is stored in unit 1's current file. (It is important to note that the new instance does not overwrite the old instance obtained from the previous execution unit; this is for ease of recovery when a task prediction is found to be incorrect). The state information associated with register R1 in the current file is updated to reflect the fact that the latest instance for R1 (as far as unit 1 is concerned) is the one produced in unit 1. Because this instance is the last possible instance for R1 in this task, it is also forwarded to unit 3 and the subsequent units.

Next, consider the execution of the second instruction of the second task. The register read access performed by this instruction is for instance $R1_2$, which is produced in the same task itself. The state information associated with register R1 in unit 1's local register file will indicate that the latest instance for R1 is $R1_2$, available in its previous file, and therefore the previous file supplies $R1_2$.

5.3. Multi-Version Register File—Detailed Working

Figure 5.3 shows the block diagram of a local register file along with the forwarding queue connecting two adjacent local register files. (This figure is intended to show only the functionality and not the implementation specifics.) As mentioned earlier, the local register file supports three instances and their associated state information for each architectural register. These instances are

---

[3] All register read accesses proceed only to the local register file of the unit that generates the register access, and each local register file is guaranteed to receive every architectural register's latest instance that logically precedes the task being executed in its unit.

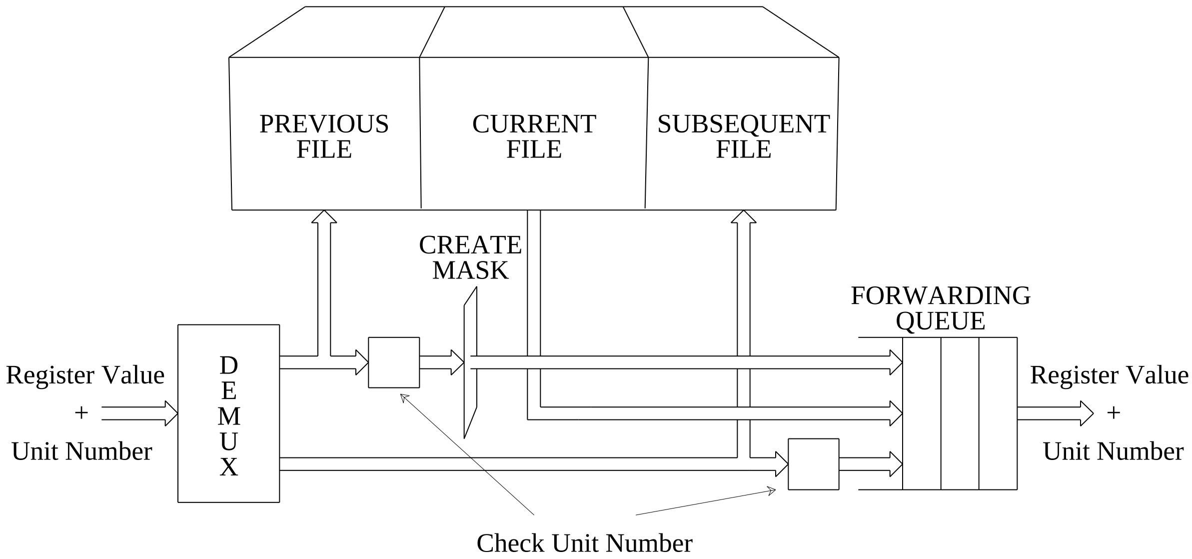


Figure 5.3: Block Diagram of a Local Register File

called *previous instance*, *current instance*, and *subsequent instance*. The state information associated with each instance identifies the logical name of an instance (previous, current, or subsequent), and contains information as to whether the instance is *valid*, *busy*, or *invalid*. The collection of previous instances and associated state information is called the *previous file*, the collection of current instances and associated state information is called the *current file*, and the collection of subsequent instances and associated state information is called the *subsequent file*.

5.3.1. Enforcing Inter-Unit Register Data Dependencies

We have discussed the local register files, and their constituent register files. Next, we shall see how exactly the register data dependencies are enforced with these mechanisms. Intra-unit register dependencies are enforced by either doing serial execution within an execution unit, or by using reservation stations (or renamed registers) with data forwarding.

Enforcing inter-unit data dependencies is more involved, as multiple tasks are executed in parallel, and the register accesses from different tasks are directed to different physical register files. The primary mechanism we use to synchronize between a register instance producer in one task and a register instance consumer in a subsequent task is by means of *busy bits* in the previous file of the consumer task. The busy bits are part of the state information associated with the previous, current, and subsequent instances in a local register file.

5.3.1.1. How are Busy Bits Set?

It is important to see how the busy bits are set. Before a task starts execution, depending on its register data dependencies with the previous tasks, the relevant busy bits of its previous file must be set. In order to execute multiple tasks in parallel, these dependencies must be known before all instructions of the previous active tasks are fetched and decoded. How is this possible? The solution is to use what we call *create masks*.

Create Mask: For a given task, the registers through which internally-created values flow out of the task and could potentially be used by subsequent tasks can be concisely expressed by a bit map called create mask. Because a multiscalar task can be an arbitrary subgraph, with arbitrary control flow within it, the create mask has to consider all possible paths through the task. Thus, the presence of a bit in a create mask does not guarantee that a new instance is created in that task for the corresponding register. Figure 5.4 shows the create mask for a task in which new instances are potentially created for registers R2 and R6, in a processor with 8 general purpose registers.

R7							R0
0	1	0	0	0	1	0	0

Figure 5.4: An Example Create Mask

Generation of Create Masks: If adequate compile-time support is available, the create mask can be generated by the compiler itself. All optimizing compilers invariably do dataflow analysis [7, 52]; the create mask is similar to the `def` variables computed by these compilers for each basic block, except that the former represents architectural registers and the latter represent variables of the source program.

Forwarding of Create Masks: When a new task is allocated to execution unit i, the task's create mask is forwarded one full round through the circular queue of execution units, starting from execution unit $i+1$ mod n, where n is the number of execution units. The create mask is forwarded one unit at a time, using the forwarding queues. When a forwarded create mask arrives at a unit, depending on whether the unit belongs to the active window or not, the ''busy bits'' of the unit's subsequent file or previous file are set. Later, in section 5.2.3.2, we will see that the same forwarding queues are also used for forwarding register values. It is of prime importance that a forwarded value for an architectural register R never overtakes in the forwarding network any create mask containing register entry R. This is because the presence of register entry R in a create mask guarantees that the correct instance for architectural register R will arrive at a *later* time. If the register value is allowed to overtake the create mask, the busy bits set by the create mask for that architectural register may never get reset, leading to deadlock. In order to prevent the occurrence of such incidents, the create masks and the register values are forwarded on a strictly first-come-first-serve basis.

The create mask captures a good deal of the information (related to register traffic) in a task in a simple and powerful way. If there were no create masks, each instruction in the large dynamic window will have to be decoded before identifying the destination register and setting the corresponding busy bit in the register file. Subsequent instructions in the overall large window, even if independent, have to wait until all previous instructions are decoded, and the busy bits of the appropriate registers are set. The advantage of having a create mask is that all the registers that are possibly written in a task are known immediately after the mask is fetched, *i.e.*, even before the entire task is fetched from the instruction cache and decoded. (As mentioned earlier, this ''decoding'' problem is a major problem in dynamically scheduled processors proposed to date.) Independent instructions from subsequent tasks can thus start execution, possibly from the next cycle onwards, and the hardware that allows this is much simpler than the hardware required to decode a large number of instructions in parallel and to compare their source and destination registers for possible conflicts.

5.3.1.2. How are Busy Bits Reset?

The busy bit of a register (in the previous file) is reset when the correct register instance reaches the previous file. The instance required by the previous file is the *last possible instance* for that register produced in a preceding task, and if multiple preceding tasks produce new instances for the register, then the required instance is the latest of these last possible instances. Therefore, the last possible register instances in a task are forwarded to the subsequent execution units. Let us see in more detail how this forwarding is done.

Forwarding of Last Possible Register Instances: When a register instance is identified to be the last possible instance for that register in that task, the result is forwarded to the subsequent execution units, one unit at a time, using the forwarding queue. A tag is attached to the forwarded result to identify the execution unit that produced the result. This tag serves two purposes: (i) it helps to determine the condition when the result has been circulated one full round through the execution units, and (ii) it helps to determine whether the result should update the previous file or to the subsequent file.

When a result from a previous unit reaches a unit i, the accompanying tag (which identifies the unit that created the result) is checked to determine if the result should be stored in the previous file or the subsequent file. If the result was generated in a preceding unit in the active window, then it is stored in the previous file; otherwise it is stored in the subsequent file. The state information associated with the selected file is updated to reflect the fact that the updated instance is valid.

The incoming unit number is checked to see if the register value has gone one full round, *i.e.*, if the incoming unit number is $(i+1) \bmod n$, where n is the number of units. If the value has not gone one full round, then the value may be forwarded to the subsequent execution unit in the next cycle if one of the following two conditions hold: (i) the update was performed on the subsequent file, or (ii) the update was performed on the previous file and the register entry does not appear in unit i's create mask. If the entry appears in the create mask, it means that the subsequent units need a different instance of that register. Since most of the register instances are used up either in the same task in which they are created or in the subsequent task, we can expect a significant reduction in the forwarding traffic because of the create mask, as shown in the empirical results of [59]. Notice also that register values from several units can simultaneously be traveling to their subsequent units in a pipelined fashion.

Identification of Last Possible Register Instances: An instance for register R can be determined to be the last possible instance for register R in a task, at the point when it is guaranteed that no new instance will be created for the register, whichever path is taken during the remainder of the execution of the task. Thus, the identification of a last possible register instance takes place either when the instruction producing the instance is executed, or when the branch that decides that the instance is the last possible is resolved. It is possible for the compiler to encode this information along with the instruction that creates the instance or along with the branch that decides that the instance is the last possible. If the use of compile-time support is not an option for generating this information (for example if object code compatibility is required), this information can be generated at run time by a special hardware the first time the tasks are encountered, and stored in a dynamic table for later reuse.

Resetting Busy Bits Set by Conservative Create Masks: As we saw in section 5.2.3.1, the create mask of a task considers all possible paths through the task, and is therefore conservative. Thus, the presence of a bit in a create mask does not guarantee that a new instance is created in that task for the corresponding register. Therefore, when a task is assigned to an execution unit *i*, the create mask forwarded to the remaining units is conservative, and may set the busy bits of registers for which it may never produce a new instance. Furthermore, unit *i* will also not forward instances of these registers from previous units, because those register entries appear in its create mask. Therefore, the create mask needs to be refined later when the branches in the task are resolved. Refining the create mask involves updating the value of the create mask, as well as forwarding any register instances (from previous units) that should have been forwarded but were not forwarded because of the original, conservative create mask. The values to be forwarded can be obtained from the previous file of unit *i*. This update can be done after each branch is resolved, or at the end of the execution of a task.

5.3.2. Register Write

When a register write access to register R is issued in unit *i*, the access is directed to unit *i*'s local register file. The value is written as the current instance of R, and the current instance is marked as *valid*. If the value written is the last possible instance for register R in the task being executed in unit *i*, then the value is also placed in unit *i*'s forwarding queue. The formal algorithm for executing a register write is given below.

```
Execute_Register_Write(R, Value)
{
        current instance of R = Value
        Mark current instance of R as Iready and valid
        if  (Value is last possible instance for R)
                Enter Value and tag i in Forwarding Queue
}
```

5.3.3. Register Read

When a register read access to register R is issued in unit *i*, the access is directed to unit *i*'s local register file. If the current instance of R is valid, then it takes the current instance. If it is busy, then the instruction waits. If it is invalid, then the previous instance is checked. If the previous instance is valid, then it is taken, and if it is busy, then the instruction waits. The formal algorithm for executing a register read is given below.

```
Execute_Register_Read(R)
{
        if  (current instance of R is valid)
                Value = current instance
        else if  (current instance of R is busy)
                wait
        else if  (previous instance of R is valid)
                Value = previous instance
        else
                wait
}
```

Figure 5.5 gives the digital logic circuit for executing a register read operation. The 3 instances corresponding to an architectural register are denoted by Instance 0, Instance 1, and Instance 2. Also, if Instance c ($c \in \{0, 1, 2\}$) is the current instance, then the previous instance is given by $(c-1)$ mod 3. From the figure, it is important to note that the extra gate delay introduced to a register read access is only 2 gate delays.

5.3.4. Committing an Execution Unit

When the execution unit at the head is committed, the unit is no longer a part of the active multiscalar window. As part of the committing process, for each architected register, among its instances present in the previous, current, and subsequent files, the latest one is renamed as the new previous instance. The latest instance among the three is identified as follows: if the subsequent instance is valid or busy, then it is the latest; else if the current instance is valid, then it is the latest; else the previous instance is the latest. An important aspect to be noted here is that to rename the instances, data needs to be moved only logically. This logical move is easily implemented by updating the state information associated with the three instances of an architected register in the execution unit. Thus, the execution unit's previous file, current file, and subsequent file are renamed on a per-register basis

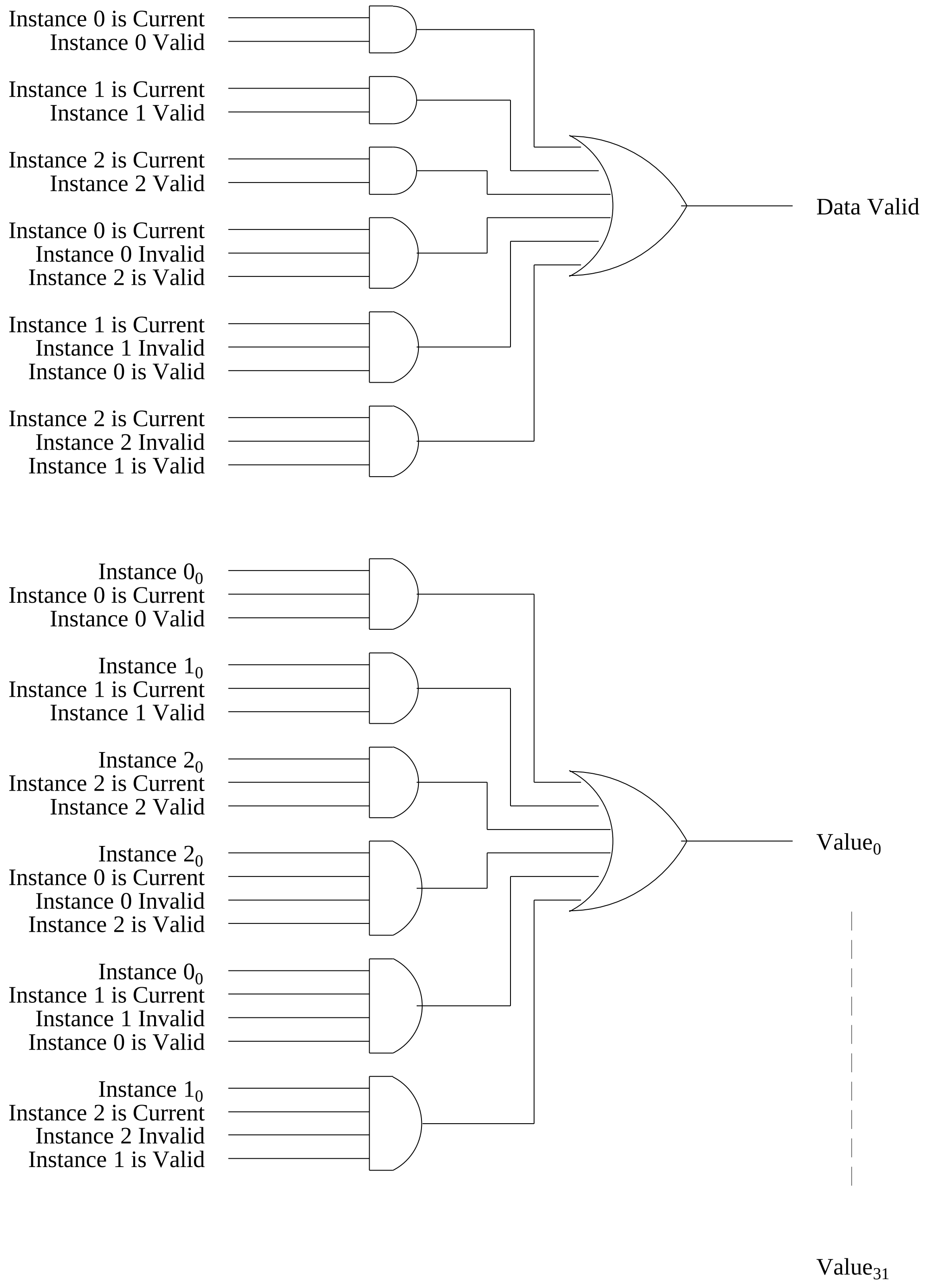


Figure 5.5: Block Diagram of Circuit for Performing Register Read Access

as in the shadow register file scheme of [142].

5.3.5. Recovery Actions

When an execution unit *i* gets squashed due to a recovery event, its create mask is forwarded one full round through the circular unit queue, one unit at a time, just like the forwarding done when a new task is allocated. When a forwarded create mask arrives at a unit *j*, depending on whether unit *j* belongs to the active window or not, the appropriate busy bits of unit *j*'s subsequent file or previous file are set. The reason for setting the busy bits again is that if unit *i* had created a new register instance and had forwarded it earlier, then that instance is no longer valid. After forwarding its create mask to the next unit, unit *i* forwards from its previous file (using the forwarding queue) those register instances whose busy bits are not set, and whose register names appear in the create mask. This step is to restore the correct values of other units' register values that were clobbered by the execution of the squashed task in unit *i*.

5.3.6. Example

It behooves us to present an example that illustrates the working of the multi-version register file in the multiscalar processor. Consider the sequence of instructions given in Table 5.1. The ‘‘Task Number’’ column in the table gives the task to which these instructions belong, and the ‘‘Unit Number’’ column gives the execution unit number in which the tasks are executed. There are 3 architectural registers, namely R0 - R2, in this example. The create mask thus consists of 3 bits. The ‘‘Execution Order’’ column shows the order in which the instructions are executed by a multiscalar processor; for simplicity of presentation, the example does not consider the execution of multiple instructions in the same cycle. For the same reason, some other functions that can be done in parallel at run time (such as completion of a task and committing the task) are also assumed to happen in separate cycles in this example.

Table 5.1: Example Sequence of Computation Operations

Label	Instruction	Task Number	Execution Unit Number	Create Mask R0R1R2	Execution Order	Result
I_0	R0 = R0 + 1	0	0	1 0 0	1	10
I_1	R2 = R0 × 4	1	1	1 0 1	2	40
I_2	R0 = R2 − 20	1	1		4	20
I_3	R1 = R2 + 20	2	2	0 1 0	3	60

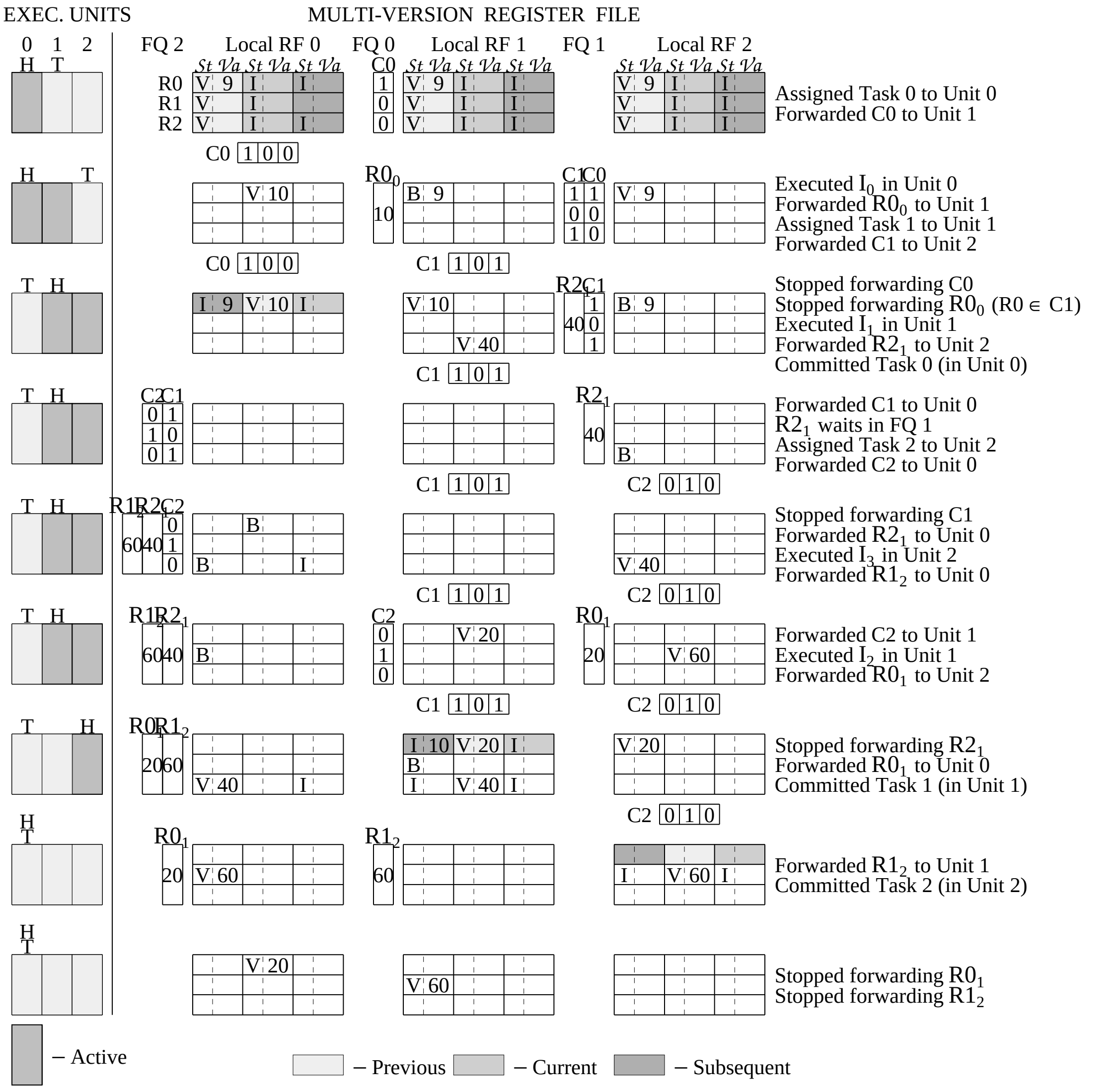


Figure 5.6: Multiscalar Active Window and Contents of the Multi-Version Register File After the Execution of Each Instruction in Table 5.1

Figure 5.6 shows the progression of the multiscalar window contents and the multi-version register file contents as the tasks are assigned and the instructions are executed. There are 9 sets of figures in Figure 5.6, one below the other, each set representing a separate clock cycle. There are 3 execution units in the figures, with the active execution units shown in dark shade. The head and tail units are marked by H and T, respectively. There are 6 columns associated with each local register file (RF) (two each for the previous file, current file, and subsequent file); the ''*St*'' columns denote state information and the ''*Va*'' columns indicate the register value. The state information associated with a register instance indicates if the instance is valid (V), busy (B), or invalid (I). Different shades are used to indicate the previous instances, current instances, and subsequent instances. A blank entry for a state field or a value field means that the content of the field is same as that in the figure immediately above it. Similarly, an unshaded register instance entry means that the instance type is same as that in the figure immediately above it. A register instance is denoted by Rx_y, where x is the register number and y is the unit in which the instance is created. A create mask is denoted by Cy, where y is the unit to which the mask's task has been allotted. The forwarding queues, FQ 0 - FQ 2, may contain forwarded create masks (in which case the value shown is in binary) or forwarded register values (in which case the value shown is in decimal).

With the description of Figure 5.6 in place, next let us go through the assignment of tasks to the execution units and the execution of instructions in the units. The first horizontal set of figures in Figure 5.6 corresponds to the cycle in which task 0 is assigned to execution unit 0. The create mask for this task (C0) is equal to binary 100, and is forwarded to the subsequent unit through the forwarding queue FQ 0. In the next cycle, execution unit 1 processes C0 and forwards it to execution unit 2, as shown in the next horizontal set of figures. Processing of C0 by unit 1 involves updating the state associated with its previous instance of register R0 from V to B in order to reflect the fact that the instance is busy. In the same cycle, instruction I_0 is executed in execution unit 0, producing $R0_0$. The current file of unit 0 is updated with the new value of R0 (namely 10), the state information updated to V, and $R0_0$ is forwarded to execution unit 1 through FQ 0. In the next cycle (c.f. third horizontal set of figures), unit 1 processes $R0_0$ by updating its previous instance of R0 to V. Had the create mask C0 arrived later than $R0_0$, then the state of the previous instance would have never changed from B to V, and task 1 would either have waited for ever for $R0_0$ or would have used the wrong value (namely 9) for R0. In the third cycle, the committing of unit 0 also takes place, as the execution of task 0 is over. Committing involves changing the state information associated with the instances of R0 in unit 0. The current instance is renamed as the new valid previous instance, the subsequent instance is renamed as the new invalid current instance, and the previous instance is renamed as the new invalid subsequent instance. The assignment of the remaining tasks and the execution of the remaining instructions can be similarly followed.

5.4. Novel Features of the Multi-Version Register File

5.4.1. Decentralized Inter-Instruction Communication

The multi-version register file maintains a separate register file for every execution unit. All register reads and writes occurring in a unit are directed only to the unit's register file. Thus the multi-version register file decentralizes the inter-instruction communication occurring in the multiscalar processor. Communication between the multiple register files are through unidirectional serial links; there is no crossbar-like connection. If an execution unit *i* needs a register instance from a previous execution unit *j*, it does not access the unit *j*'s local register file; rather, the required instance is guaranteed to arrive at unit *i*'s local register file. Decentralization of the register file drastically reduces the number of read ports and write ports each physical register file needs to have.

5.4.2. Exploitation of Communication Localities

Not only does the multi-version register file decentralize the inter-instruction communication, but also does it exploit localities of communication. Because a significant number of register instances are used up soon after they are created [59], most of the inter-instruction communication occurring in the multiscalar processor falls under intra-task communication, which can be done in parallel in multiple units. Empirical results presented in [59] show that most of the register operands are generated either in the same task in which they are used or in the preceding task. Similarly, most of the register instances need not be forwarded to the subsequent units, and even among the forwarded instances, most of the instances are forwarded to at most one unit. Those results show that the multi-version register file exploits the localities of inter-instruction communication present in programs.

5.4.3. Register Renaming

The multi-version register file provides renaming between the registers specified in different tasks, by directing register references from different tasks to different physical registers. Apart from this global register renaming, each execution unit can have its own independent register renaming also if desired; any register renaming done locally within a unit is independent of the global renaming done by the multi-version register file.

5.4.4. Fast Recovery

The multi-version register file maintains precise state at task boundaries, with the help of the previous files. Because of this feature, recovery actions in times of incorrect task prediction and other untoward incidents are greatly simplified.

5.5. Summary

The inter-instruction communication mechanism of the multiscalar processor must provide high bandwidth and good hardware support for speculative execution. Just like the other resources described in chapter 4, the inter-instruction communication in the multiscalar processor is also broken into an intra-unit part and an inter-unit part. We developed a decentralized inter-instruction communication mechanism for the multiscalar processor. The central idea of the decentralized register file scheme is to have multiple versions of the architected register file. The multiple versions provide a platform to store the speculative register values produced in the execution units. Thus, the intra-unit communication is handled by the local register files.

The inter-unit communication is also carried out in a distributed manner, with the help of a *ring-type forwarding network*, which forwards register values from one execution unit to the next unit. Only the last updates to the registers in a task are passed on to the subsequent units, and most of these last updates need not be propagated beyond one or two units. Therefore, much of the register communication occurring in the multiscalar processor is ‘‘localized’’. Thus we exploit the temporal locality of usage of register values to design a good decentralized register file structure that ties well with the multiscalar execution model. The multiple versions also allow precise state to be maintained at inter-unit boundaries, which facilitates recovery actions in times of incorrect task prediction. Having multiple versions of the register file has the benevolent side-effect that the number of read ports and write ports of each register file is much less than that required with a single centralized register file.

Chapter 6. Data Memory System

How to perform out-of-order execution of multiple memory references per cycle?

This chapter deals with the data memory system for the multiscalar processor. The memory system of the multiscalar processor has to deal with several aspects. First, it should provide sufficient bandwidth for the processor to execute multiple memory references per cycle. Second, memory references from multiple tasks are executed as and when the references are ready, without any regard to their original program order. To guarantee correctness of execution when references are executed before stores that precede them in the sequential instruction stream, memory references need to be disambiguated. Finally, most of the execution in the multiscalar processor is speculative in nature. To support speculative execution of memory references, there should be a hardware platform to hold the values of speculative stores (until they are guaranteed to commit), from where they have to be forwarded to subsequent load requests. To support the first aspect, we use multi-ported (interleaved) non-blocking cache that can service multiple requests per cycle and overlap the service of misses. To support the second and third aspects, we use a hardware mechanism, called *Address Resolution Buffer (ARB)*. The ARB supports the following features: (i) dynamic memory disambiguation in a decentralized manner, (ii) multiple memory references per cycle, (iii) out-of-order execution of memory references, (iv) dynamically unresolved loads and stores, (v) speculative loads and stores, and (vi) memory renaming.

This chapter is organized as follows. Section 6.1 describes the requirements of the multiscalar processor's data memory system. Section 6.2 deals with the design of a data memory system to support the data bandwidth demands of the multiscalar processor. Section 6.3 describes existing dynamic disambiguation techniques. Section 6.4 describes the ARB and its working for the multiscalar processor. Section 6.5 describes the novel aspects of the ARB. Section 6.6 describes set-associative ARBs and other extensions to the ARB, such as those for handling variable data sizes. Section 6.7 summarizes the chapter.

6.1. Requirements of the Multiscalar Data Memory System

The requirements of the multiscalar data memory system can be grouped under five attributes: (i) low latency, (ii) high bandwidth, (iii) support for speculative execution of memory references, (iv) dynamic memory disambiguation, and (v) dynamically unresolved memory references. The first criterion is very obvious, and is universally applicable to any processing paradigm for exploiting irregular parallelism. The rest of the criteria are explained in detail below.

6.1.1. High Bandwidth

The data bandwidth demands of the multiscalar processor are much higher than those of single-issue processors because: (i) fewer clock cycles are taken to execute the program and service the same number of ‘‘useful’’ data references and (ii) in this fewer cycles, many additional data references may be generated due to incorrect speculative execution. Perhaps the data memory is the most heavily demanded resource after the inter-operation communication mechanism, and is likely to be a critical resource. The instruction issue rate that the multiscalar processor can sustain is limited to $\frac{(1-f_S)\,BW_M}{f_M}$ instructions per cycle, where f_S is the fraction of instructions that are squashed, BW_M is the average bandwidth that the data memory can supply, and f_M is the fraction of all instructions that are memory references. Clearly, any improvements in the execution strategy will be worthwhile only if they are accompanied by a commensurate increase in the data memory bandwidth.

6.1.2. Support for Speculative Execution

The importance of speculative execution in the multiscalar processor can never be overemphasized. While performing speculative execution, the value of a speculatively executed store operation can be allowed to proceed to the memory system only when it is guaranteed to commit; otherwise the old memory value will be lost, complicating recovery procedures in times of incorrect task prediction. Nevertheless, succeeding loads (from speculatively executed code) to the same memory location do require the new, uncommitted value, and not the old value. Thus, the data memory system of the multiscalar processor has to provide some means of forwarding uncommitted memory values to subsequent loads, and values have to be written to the memory system in the order given by the sequential semantics of the program.

6.1.3. Support for Dynamic Memory Disambiguation

As explained in chapter 3, memory references from different execution units of the multiscalar processor are executed out-of-order, with no regard paid to their original program order. However, care has to be taken that no two memory references (at least one of which is a store) to the same location are reordered, lest the execution becomes incorrect. Therefore, the multiscalar processor needs a good dynamic disambiguation mechanism. Dynamic disambiguation techniques for dynamically scheduled processors involve comparing the addresses of all loads and stores in the active instruction window; existing hardware mechanisms do this by means of associative compares. When the window is large, the associative compare becomes extremely complex. The latency introduced by the disambiguation hardware could then significantly increase the execution latency of memory references. The problem gets exacerbated when multiple references are issued in a cycle, forcing the disambiguation mechanism to perform associative compares of multiple loads and stores. Thus, we need a better, decentralized hardware mechanism for performing dynamic memory disambiguation in the multiscalar processor.

6.1.4. Support for Dynamically Unresolved Memory References

As discussed in section 2.3.3.2, to extract lots of parallelism at run time from a large instruction window, an ILP processor should support dynamically unresolved references. That is, the disambiguation mechanism should allow the execution of memory references before disambiguating them from the preceding stores, or even before the addresses of the preceding stores are known. If and when an unresolved reference violates any memory data dependencies, the multiscalar processor can recover from the incorrect execution by using the recovery facility already provided to incorporate speculative execution. We expect that a good compiler would keep soon-to-be-used values in registers, and therefore most of the time, the dynamically unresolved loads will fetch the correct values from the memory system.

6.2. High-Bandwidth Data Memory System

Traditional data caches are single-ported blocking caches in which at most one memory request can be sent to the cache every cycle, and at most one miss can be serviced at any one time. The disadvantages of a single ported, blocking cache are the bandwidth degradation due to the serial handling of misses, and the bandwidth limitation due to a single port.

For providing a high bandwidth data memory system for the multiscalar processor, we use the multi-ported non-blocking cache that we had proposed earlier for superscalar processors [148]. The multiple ports improve the bandwidth to more than one request per cycle, and the non-blocking feature helps to reduce the bandwidth degradation due to misses. Figure 6.1 shows the block diagram of the multiscalar processor's data memory system. The multiple ports are implemented by using multiple cache banks. The cache blocks are interleaved amongst the multiple banks, with a cache block entirely present in a single cache bank. A straightforward way to do the interleaving of cache blocks is to use low-order interleaving of block addresses. Other interleaving schemes, such as those described in [128, 145] could be used, and need further study.

One potential drawback of a multi-banked cache is the additional delay introduced by the interconnection network (ICN) from the execution units to the multiple banks (see Figure 6.1). Passing through this interconnect to get to the cache can potentially degrade the latency of cache hits. It is possible to pipeline the references, however, so that passage through the interconnect is just an extra stage in the execution of a memory reference. Section 9.2 further addresses this issue.

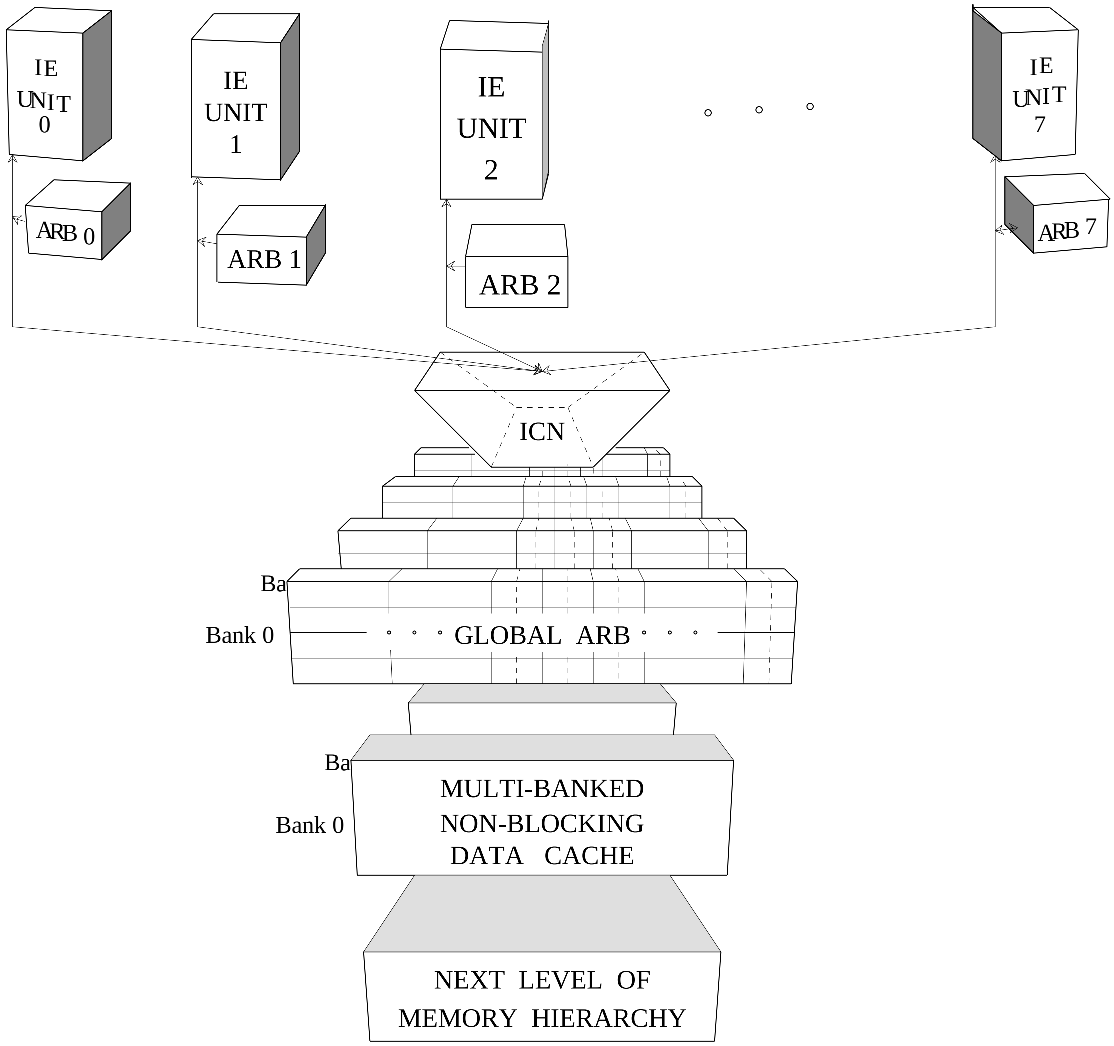


Figure 6.1: A Multi-Banked Non-Blocking Cache for an 8-Unit Multiscalar Processor

6.3. Existing Dynamic Disambiguation Techniques

Having discussed ways of providing a high bandwidth data memory system for the multiscalar processor, the next issue to be addressed is enforcing the dependencies through memory, *i.e.*, doing dynamic disambiguation of memory references, which may get reordered in the multiscalar processor. Before that, let us review the published techniques for performing dynamic memory disambiguation. The published techniques can be broadly classified into 2 categories: (i) techniques for use in control-driven execution models (which support statically unresolved loads), and (ii) techniques for use in data-driven execution models (which support out-of-order execution of loads and stores). Below we review some schemes from each of the two categories.

6.3.1. Techniques for Control-Driven Execution

Why would dynamic disambiguation be needed in a processor following control-driven execution? The answer is that in order to overcome the limitations imposed by inadequate static memory disambiguation on the static extraction of parallelism, some processing paradigms allow the compiler to reorder statically unresolved references, with checks made at run time to determine if any dependencies are violated by the static code motions. We shall review the techniques proposed for dynamic disambiguation of such statically unresolved memory references that are reordered at compile time.

Run-Time Disambiguation (RTD): The run-time disambiguation (RTD) scheme proposed by Nicolau [108] adds extra code in the form of explicit address comparison and conditional branch instructions to do run-time checking of statically unresolved references. Corrective action is taken when a violation is detected. This corrective action typically involves executing special recovery code to restore state and restart execution. Notice that RTD does not use any special hardware support, although it performs disambiguation at run time. By deferring the checks to run time, RTD allows general static code movement across ambiguous stores. Although static disambiguation can help reduce the extra code to be inserted, the extra code can still be large when aggressive code reordering is performed.

Non-Architected Registers: Silberman and Ebcioglu [132] describe a technique for supporting statically unresolved loads and multiprocessor consistency in the context of object code migration from CISC to higher performance platforms, such as the superscalars and the VLIWs. They propose to use *non-architected registers* to temporarily store the results of statically unresolved loads. Each non-architected register has an address tag field to hold the memory address from which the register was loaded. Copy operations copy the address tag from the source to the destination without modification. When a store is executed, its address is associatively checked against the address tags of all non-architected registers of its processor and other processors in the same multiprocessor group. For each register, if a match exists, meaning that an incorrect unresolved load has loaded this register, the register's exception tag is set. If and when a non-architected register is committed (copied) to the actual architected target register of the load instruction, and it has the exception tag set, an exception occurs. Execution then continues by re-executing the offending load. Notice that the committing copy operation has to be done before the execution of any ambiguous store that succeeds the unresolved load in the original program sequence. As far as possible, these copy instructions are scheduled in unoccupied slots in the VLIW instruction sequence. The disadvantages of this scheme are (i) a copy operation is needed for every statically unresolved load, and (ii) for every store operation, the hardware has to perform an associative search among the address tags of all non-architected registers.

Memory Conflict Buffer (MCB): The MCB scheme proposed by Chen, *et al* [31] is similar to the scheme discussed above. The difference is that it has no non-architected registers, and instead an address tag is associated with every general purpose register. Statically unresolved loads are specified by a special opcode called *preload*, instead of by the type of target register used. Stores will check for address conflicts among all valid address tags, and set a conflict bit associated with the appropriate general purpose register if there is an address match. A *check* instruction is used in place of the committing *copy* operation. Recovery actions are taken by means of correction code supplied by the compiler. The disadvantages with this scheme are similar to those of the previous scheme, namely, (i) a check instruction is needed for every ambiguous load-store pair, and (ii) when a store is performed, in the worst case, an associative compare has to be performed among all general purpose registers, which in a VLIW processor can be as much as 128 registers [100].

6.3.2. Techniques for Data-Driven Execution

In processors following data-driven execution, if loads are executed after proper disambiguation, then to determine the executable loads in a given cycle, the hardware typically compares the addresses of all unexecuted loads in the instruction window with the addresses of all unexecuted stores that precede them in the instruction window. On the other hand, if loads are executed before proper disambiguation, then each time a store is executed, the hardware typically compares the store address with the addresses of all executed loads that succeed it in the instruction window. In either case, there is a wide associative search[4], and this (centralized) associative search for potential memory hazards limits the dynamic instruction window size, and hence the parallelism that can be exploited. Existing hardware disambiguation schemes are plagued by this hardware complexity, which can lead to unacceptable cycle times. Below we review some of the existing schemes for run-time disambiguation in data-driven execution processors.

Store Queue: The store queue is the disambiguation mechanism used in the IBM System/360 Model

---

[4] In the former, every cycle, the hardware checks if any of the loads in the instruction window can be executed. The maximum number of unexecuted loads in the instruction window and the maximum number of unexecuted stores are both W, where W is the instruction window size. Thus, to handle the worst case, it requires hardware for $\sum_{i=1}^{i=W} i-1 = \frac{W(W-1)}{2}$ compares. In the latter, every cycle, the hardware need only check if the stores executed in that cycle conflict with any of the loads executed prematurely. Thus, to handle the worst case, the latter requires hardware for $\sum_{i=1}^{i=S} W-i = \frac{S(2W-S-1)}{2}$ compares, where S is the maximum number of stores executed per cycle.

91 [15, 22] and Astronautics ZS-1 [139]. In this scheme, the stores are allowed to wait in a queue until their store values are ready. Loads are given higher priority than stores, and are allowed to pass pending stores waiting in the store queue, as long as they do not pass a store to the same address. Loads entering the memory system have their addresses compared with the addresses in the store queue, and if there is an address match the load is not allowed to proceed. (This compare can be done after fetching the data from memory too, as in the IBM System/370 Model 168.) Emma, *et al* [50] proposed a similar mechanism, which in addition to executing loads before preceding stores, supports sequential consistency in a multiprocessor system by allowing each processor to monitor the stores made by the remaining processors. Memory disambiguation with a store queue requires centralized interlock control and associative compare of the load and store addresses in the instruction window, which can be too complex when the window is big as in the multiscalar processor. The store queue also does not support dynamically unresolved references, rendering it as a poor candidate for use in the multiscalar processor.

Stunt Box: This is a mechanism used in the CDC 6600 [152] for dynamically reordering the memory references that are waiting due to memory bank conflicts. The way out-of-order loads and stores to the same memory location are prevented is by disallowing loads and stores to be present at the same time in the stunt box. The stunt box is simpler than the store queue, allows only limited reordering of memory references, and is even more restrictive than the store queue.

Dependency Matrix: This is a scheme proposed by Patt, *et al* in connection with the HPS restricted dataflow architecture [115], which performs dynamic code reordering and dynamic disambiguation. In this scheme, a dependency matrix is provided for relating each memory reference to every other reference in the active instruction window. Each load is assigned a row in the dependency matrix, which is essentially a bitmap; the bits corresponding to preceding stores with unknown addresses will be set to 1. As store addresses become known, bits are cleared in the bitmap until either all bits have been cleared (in which case the load can be sent to the data cache / main memory) or a single one remains and that one corresponds to a store whose address and store value are known (in which case the store value can be forwarded to the load). Patt, *et al* acknowledge that this scheme requires a high level of intercommunication between different parts of the memory unit. Also notice that a load operation has to wait until the addresses of all preceding stores are known, *i.e.*, dynamically unresolved loads and stores are not supported.

6.4. Address Resolution Buffer (ARB)

Looking at existing solutions for dynamic memory disambiguation in data-driven execution models, all of them require wide associative compares or restrict the scope of memory reference reordering, and run counter to the multiscalar theme of decentralization of critical resources. None of the dynamic disambiguation schemes described support dynamically unresolved references. We devised the Address Resolution Buffer (ARB) to overcome all these limitations and to support the speculative execution of memory references. In this section, we describe its working for the multiscalar processor. The appendix describes how the ARB can be used in other processing models such as the superscalar processors, the VLIW processors, and the multiprocessors.

The first step we adopt to decentralize the memory disambiguation process in the multiscalar processor is to divide it into two parts: (i) *intra-unit memory disambiguation* and (ii) *inter-unit memory disambiguation*. The former is to guarantee that out-of-order load-store accesses and store-store accesses do not occur to the same address from an execution unit. The latter is to guarantee that out-of-order load-store accesses and store-store accesses do not occur to the same address from multiple execution units, given that accesses to an address are properly ordered within each unit. We shall first discuss inter-unit memory disambiguation (given that the references are properly ordered within each unit).

6.4.1. Basic Idea of ARB

The basic idea behind the ARB is to allow out-of-order issue and execution of memory references, but provide a hardware platform to order the references sequentially. To record the original order of the memory references, each memory reference is identified by its execution unit number. Memory references from different units are allowed to proceed to the memory system in any order (as and when they are ready to proceed). Having allowed the out-of-order execution of memory references from multiple units, all that is needed now is a hardware structure that efficiently detects out-of-sequence load-store and store-store references to the same memory location as and when they occur.

6.4.2. ARB Hardware Structure

The ARB is the hardware mechanism we use for detecting out-of-order load-store and store-store references from different units to the same memory location. It is a special cache-like structure, with provision to hold information relevant to the loads and stores executed from the active execution units. The ARB is interleaved (based on the memory address) to permit multiple memory accesses per cycle. The degree of interleaving is commensurate with the maximum number of memory requests that can be executed per cycle from the entire processor. Each ARB bank has several row entries, and each row entry has an *address* field for entering a memory address. The rest of the row is divided into as many *stages* as there are execution units in the multiscalar processor. Each ARB stage has a *load bit*, a *store bit*, and a *value* field. The load bit is used for indicating if a load has been executed from the corresponding execution unit to the address in the address field, and the store bit is for indicating if a store has been executed from that execution unit to the address in the address field. The value field is used to store one memory value. The ARB stages can be thought of as ‘‘progressing in sequential order’’, and have a one-to-one correspondence to the execution units of the multiscalar processor. The ARB stages are also logically configured as a circular queue, with the same *head* pointer and a *tail* pointer as the execution unit queue. The active ARB stages, the ones from the head pointer to the tail pointer, together constitute the *active ARB window*, and corresponds to the active multiscalar window.

Figure 6.2 shows the block diagram of a 4-way interleaved 6-stage ARB. In this figure, the head and tail pointers point to stages 1 and 5 respectively, and the active ARB window includes stages 1 through 4, corresponding to the four execution units 1 through 4. A load has been executed from unit 2 to address 2000, and its ARB entry is in bank 0. Similarly, a store has been executed from unit 3 to address 2048 (ARB bank 1), and the store value is 10.

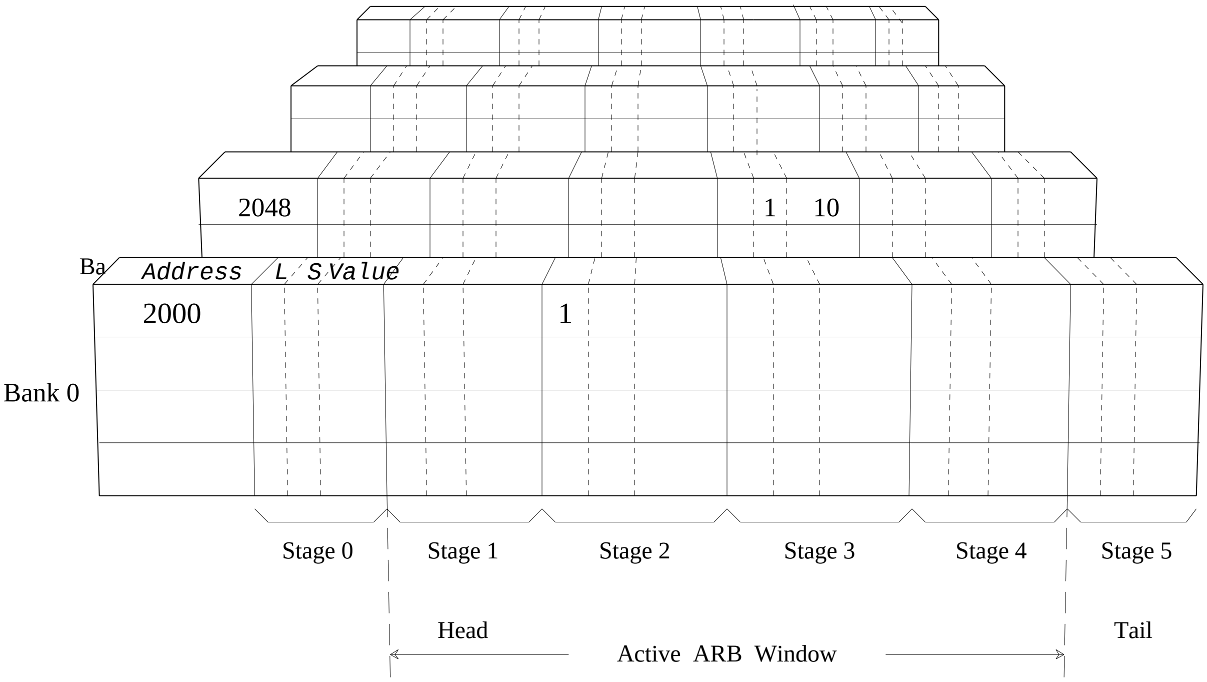


Figure 6.2: A 4-Way Interleaved, 6-stage ARB

6.4.3. Handling of Loads and Stores

When a load is executed from execution unit *i*, first the ARB bank is determined based on the load address. The address fields of the entries in this ARB bank are then associatively checked[5] to see if the load address is already present in the ARB. If the address is present in the ARB bank, then a check is made in its row entry to see if an earlier store has been executed to the same address from a preceding unit in the active window. If so, the store from the closest unit is determined, and the value stored in its value field is forwarded to the load's destination register. Otherwise, the load request is forwarded to the data cache / main memory. Thus, a load need not proceed to the data cache / main memory if a preceding store has already been executed from the active window[6]. If the address is not present in the ARB bank, a free ARB row entry is allotted to the new address; if no free entry is available, then special recovery action (c.f. Section 6.4.4) is taken. The load bit of stage *i* of the ARB entry is set to 1 to reflect the fact that the load has been executed from execution unit *i*. Notice that when multiple memory requests need to access the same ARB bank in a cycle, the one closest to the ARB head pointer is allowed to proceed, and the others are executed in the subsequent cycles. The formal algorithm for executing a load is given below.

---

[5] It deserves emphasis that this associative search is only within a single ARB bank and not within the entire ARB, and the search is for a single key. Later, in section 6.6.2, we discuss set-associative and direct-mapped ARBs, which further cut down the associative search.

[6] It is worthwhile to point out that in order for the ARB to be not in the critical path when a load ''misses'' in the ARB, load requests can be sent to both the ARB and the data cache simultaneously; if a value is obtained from the ARB, the value obtained from the data cache can be discarded.

```
Execute_Load(Addr, Sequence)
{
        Determine the ARB Bank based on Addr
        Associatively check the Bank entries for Addr
        if  (Addr not present)
        {
                if  (Bank full)
                        Recovery_Bank_Full(Sequence)
                Allocate new row entry
                Enter Addr in address field of new row entry
        }
        else
        {
                Check in the row entry if a logically preceding store has been executed to Addr
                if  (preceding store executed to Addr)
                {
                        Determine nearest such store
                        Forward the value from its value field
                }
                else
                        Forward load request to data cache / main memory
        }
        Set row entry’s load bit of stage Sequence to 1
}
```

When a store is executed from execution unit *i*, the ARB bank is determined, and a similar check is performed within the bank to see if the store address is already present in the ARB. If not, an ARB row entry is allotted to the new address. The store bit of stage *i* of the ARB row entry is set to 1 to reflect the fact that the store has been executed from unit *i*. The value to be stored is deposited in the row entry’s value field for stage *i*. If the memory address was already present in the ARB bank, then a check is also performed in the active ARB window to see if any load has already been performed to the same address from a succeeding unit, and there are no intervening stores in between. If so, recovery action is initiated. Such recovery action might involve, for instance, nullifying the execution of the tasks in the execution units including and beyond the closest incorrect load, and restaring their execution. The formal algorithm for executing a store is given below.

```
Execute_Store(Addr, Value, Sequence)
{
        Determine the ARB Bank based on Addr
        Associatively check the Bank entries for Addr
        if  (Addr not present)
        {
                if  (Bank full)
                        Recovery_Bank_Full(Sequence)
                Allocate new row entry
                Enter Addr in address field of new row entry
        }
        else
        {
                Check in the row entry if a logically succeeding load has been executed to Addr,
                                        with no intervening stores
                if  (later load with no intervening stores)
                {
                        Determine nearest such load
                        Squash its unit
                }
        }
        Enter Value in the value field of stage Sequence of row entry
        Set row entry's store bit of stage Sequence to 1
}
```

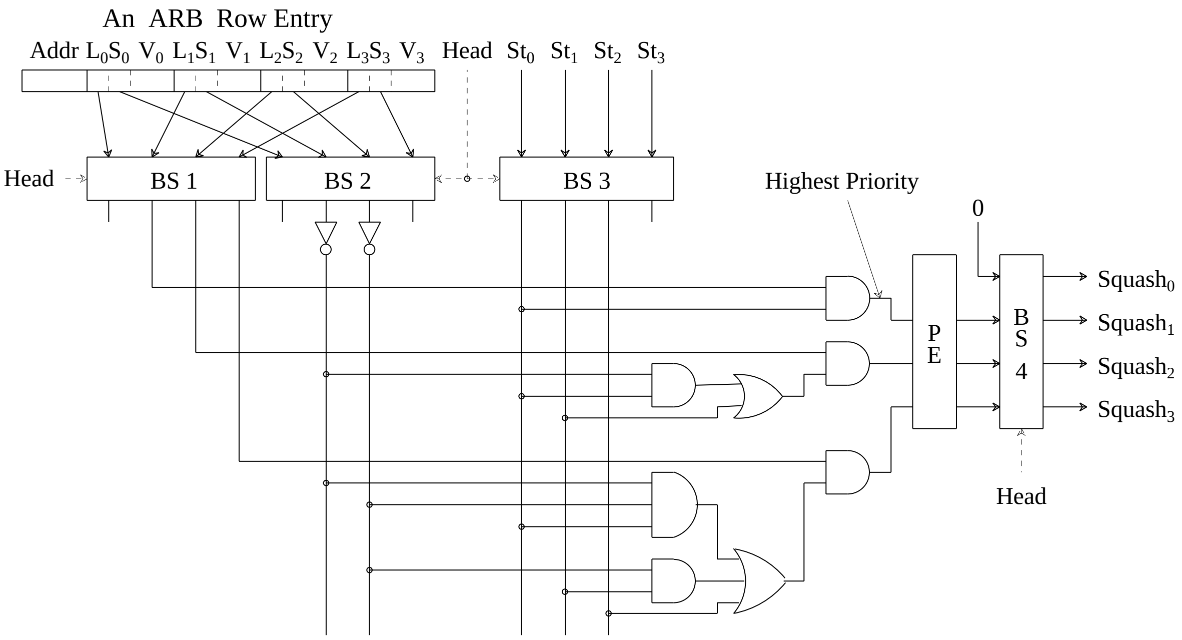
The logic for detecting out-of-order load-store pairs to the same memory location, and issuing the appropriate squash signal is given below.

$$\mathrm{Squash}_i \;=\; \mathrm{L}_i \wedge \left(\bigvee_{j=head}^{j=i-1} \left(\mathrm{St}_j \bigwedge_{k=j+1}^{k=i-1} \overline{\mathrm{S}}_k \right) \right)$$

where, j varies as $\begin{cases} head \leq j < i & \text{if } head \leq i \\ head \leq j \leq n-1, \ \ 0 \leq j < i & \text{if } head > i \end{cases}$

and k varies as $\begin{cases} j < k < i & \text{if } j < i \\ j < k \leq n-1, \ \ 0 \leq k < i & \text{if } j > i \end{cases}$

When multiple squash signals are generated, the one closest to the head is selected. Figure 6.3 gives the block diagram of the circuit for generating the squash signals in a 4-unit multiscalar processor.



L_i – Load Mark for stage i

S_i – Store Mark for stage i

V_i – Value field for stage i

St_i – Store to stage i

BS – Barrel Shifter

PE – Priority Encoder

Figure 6.3: Block Diagram of Circuit for Detecting Out-of-sequence Memory Accesses in a 4-stage ARB

6.4.4. Reclaiming the ARB Entries

When the processor retires the execution unit at the head pointer, any load marks or store marks in the corresponding ARB stage are erased immediately. Clearing of the load marks and store marks can be easily done by clearing the load bit and store bit columns corresponding to the stage at the head in a single cycle. The store values pertaining to the stores executed in that stage are also forwarded to the data cache / main memory. In a given cycle, it is possible to commit as many stores as the number of write ports to the data cache / main memory. If there are multiple store values in a stage, this forwarding could cause a traffic burst, preventing that ARB stage from being reused until all the store values have been forwarded. One simple solution to alleviate this problem is to use a write buffer to store all the values that have to be forwarded from a stage. Another solution is to forward the values leisurely, but have more physical ARB stages than the number of multiscalar units, and use some sort of ARB stage renaming, *i.e.*, by marking the unavailable ARB stages, and by dynamically changing the mapping between the execution units and the ARB stages.

When recovery actions are initiated due to the detection of out-of-sequence accesses to a memory location or detection of incorrect speculative execution, the multiscalar tail pointer is appropriately moved backwards, and the load mark and store mark columns corresponding to the stages stepped over by the tail pointer are also cleared immediately just like the clearing done when the head unit is committed. (This clearing can be done leisurely also; the only restriction is that it should be done before the execution unit corresponding to the ARB stage is reassigned a task.) An ARB row is reclaimed for reuse when all the load and store bits associated with the row are cleared.

When a memory reference is executed from unit *i* and no row entry is obtained because the relevant ARB bank is full (with other addresses), then special recovery actions are taken. The ARB bank is checked to see if there are any address entries in which all loads and stores have been executed from units succeeding unit *i* in the active ARB window. If so, one of those entries is selected, and the execution units including and beyond the reference corresponding to the sequentially oldest load mark or store mark in that entry are discarded. This discarding action will enable that row entry to be reclaimed, and be subsequently allotted to the memory reference occurring in unit *i*. If no ARB row entry is found in which the oldest executed reference is from a succeeding unit, then the memory reference at unit *i* is stalled until (i) an ARB row entry becomes free, or (ii) this reference is able to evict the row entry corresponding to another memory address. One of these two events is guaranteed to occur, because references only wait (if at all) for previous units to be committed. Thus, by honoring the sequential program order in evicting the ARB row entries, deadlocks are prevented. The formal algorithm for handling the situation when an ARB bank is full is given below.

```
Recovery_Bank_Full(Sequence)
{
        Check in the ARB Bank for entries with no load and store marks before stage Sequence
        if  (there is any such entry)
        {
                Determine entry whose earliest load/store mark is closest to ARB tail
                Squash the unit corresponding to its earliest load/store mark
        }
        else
                exit        /* The request has to be re-tried, possibly in the next cycle */
}
```

6.4.5. Example

The concepts of ARB can be best understood by going through an example. Consider the sequence of loads and stores in Table 6.1, which form a part of a sequential piece of code (instructions other than memory references are not used in the example for the purpose of clarity). The ‘‘Unit Number’’ column in the table gives the numbers of the execution units in which these memory references are executed; for simplicity of presentation, we assume each reference to belong to a different task so that no intra-task disambiguation is required. Each task is executed on a different execution unit. The ‘‘Addr.’’ column gives the addresses of the memory references in a correct execution of the program. The ‘‘Store Value’’ column gives the values stored by the two store instructions. The ‘‘Exec. Order’’ column shows the order in which the loads and stores are executed by the processor; for simplicity of presentation, the example does not consider the execution of multiple references in the same cycle. Figure 6.4 shows the progression of the multiscalar window contents and the ARB contents as the memory references are executed. There are 7 sets of figures in Figure 6.4, one below the other, each set representing a separate clock cycle. There are 4 execution units in the figures, with the active execution units shown in dark shade. The head and tail units are marked by H and T, respectively. The number of execution units (ARB stages as well) is 4 in the figure, and the ARB has only a single bank. Blank entries in the ARB indicate the presence of irrelevant data.

Assume that all tasks have been assigned to the execution units, as shown by the shaded units in the first figure. Let us go through the execution of one memory reference. The first figure in Figure 6.4 depicts the execution of the load from unit 2. This load is to address 100, and because there is no ARB row entry for address 100, a new ARB row entry is allotted for address 100. The load bit of unit 2 of this row entry is set to 1. The load request is forwarded to the memory, because no store has been executed to address 100 from the active instruction window. Notice that an earlier store (from unit 0) is pending to the same address; so the value returned from memory (say 140) is incorrect. The execution of the remaining memory references can be similarly followed.

6.4.6. Two-Level Hierarchical ARB

Now let us address intra-unit disambiguation. If in-order execution is performed within each unit, then the memory references are also executed in program order, and there is no need to do intra-group memory disambiguation. If out-of-order execution is performed within each unit, then the solution is more involved, and warrants further discussion.

To support out-of-order execution of memory references in the multiscalar execution units, we propose a two-level hierarchical ARB structure. At the top level of the hierarchy, there is a single global ARB for performing inter-unit memory disambiguation as before. At the bottom level of the hierarchy, there is a local ARB corresponding to each multiscalar unit, for carrying out intra-unit memory disambiguation. The local ARB functions the same way as the global ARB. Except for the

Table 6.1: Example Sequence of Loads and Stores

Code	Task No.	Unit No.	Addr.	Store Value	Exec. Order	Remarks
STORE R2, 0(R1)	0	0	100	120	4	
STORE R4, 0(R2)	1	1	120	50	2	
LOAD R3,−20(R2)	2	2	100		1	Unresolved load; Fetches incorrect value 140
LOAD R5, 0(R3)	3	3	120		3	

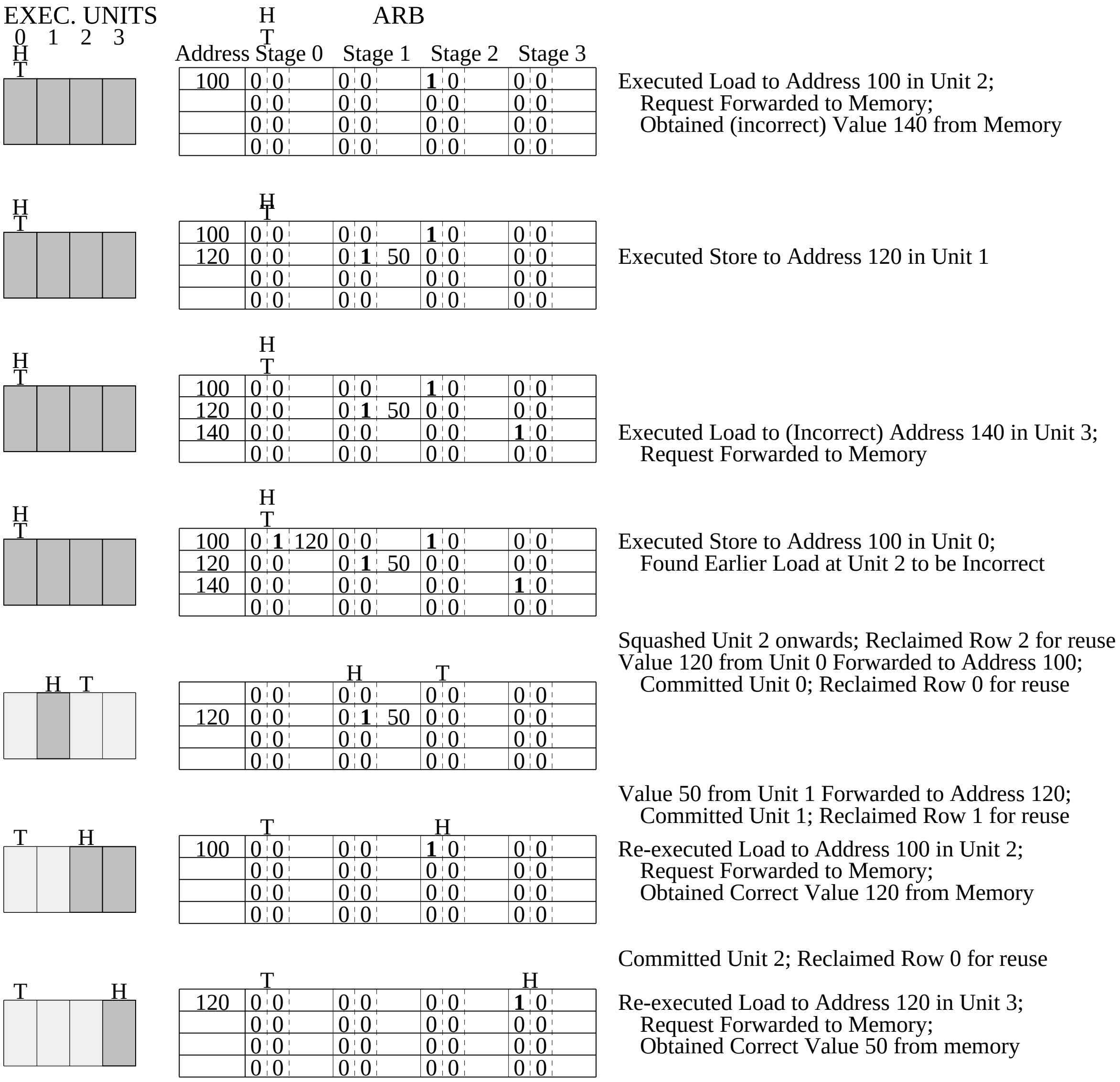


Figure 6.4: Multiscalar Active Window and Contents of the ARB After the Execution of Each Memory Reference in Table 6.1

reason of reducing the complexity of the associative search for an address entry, the degree of interleaving of the local ARBs need not be greater than the maximum number of memory requests issued in a cycle from a multiscalar unit.

Figure 6.5 shows the block diagram of a two-level hierarchical ARB. In the figure, the global ARB has 6 stages, which are interleaved into 4 banks as before. Each ARB bank can hold up to 4 address entries. The active multiscalar window consists of units 1 through 4. Each unit has an 8-stage local ARB, which is not interleaved, and can store up to 4 address entries.

The local ARBs handle the loads and stores from their respective multiscalar units much the same way as the global ARB. The only differences are: (i) when a load ‘‘misses’’ in the local ARB, instead of forwarding the request to the data cache / main memory, it is forwarded to the global ARB, (ii) when a store is executed, the store request is sent to both the local ARB and the global ARB. When the multiscalar unit at the head is retired, all load and store marks in its local ARB and the head stage of the global ARB are erased immediately. Also, all store values stored in that stage are forwarded to the data cache / main memory.

6.5. Novel Features of the ARB

The ARB performs dynamic memory disambiguation, and allows loads and stores to be executed out-of-order with respect to loads and stores from preceding execution units. Furthermore, it allows multiple memory references to be issued per cycle. It uses the concept of interleaving to decentralize the disambiguation hardware, and thereby reduce the associative search involved in dynamic memory disambiguation. Interleaving also helps to support multiple memory references per cycle. Besides these features, the ARB supports the following novel features.

6.5.1. Speculative Loads and Stores

The ARB allows loads and stores to be issued from speculatively executed code. It provides a good hardware platform for storing speculative store values, and correctly forwards them to

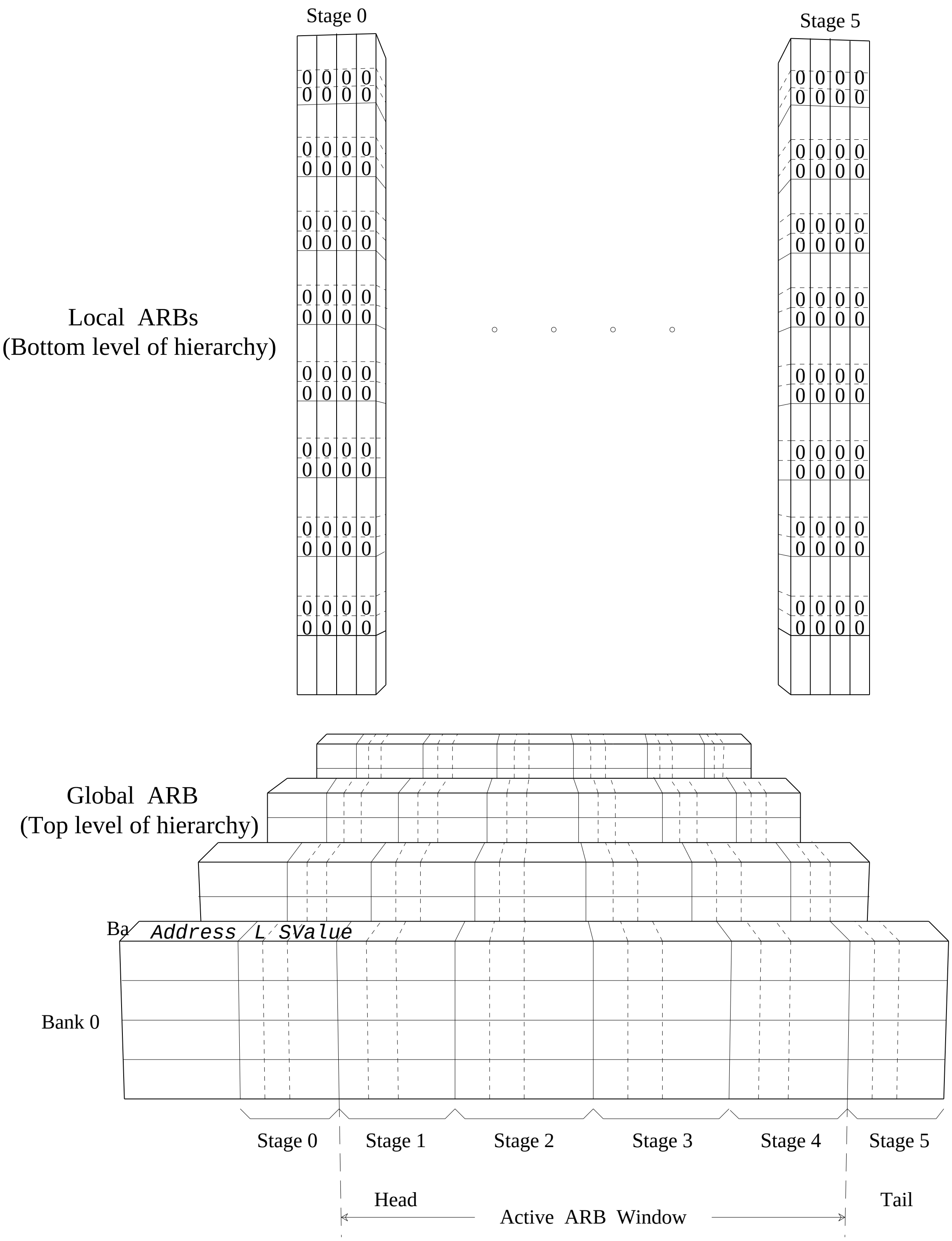


Figure 6.5: A Two-Level Hierarchical ARB

subsequent speculative loads. Moreover, the speculative store values are not forwarded to the memory system until the stores are guaranteed to commit. Recovery operations are straightforward, because they involve only a movement of the tail pointer and the clearing of the appropriate load bit and store bit columns; the incorrect speculative values are automatically discarded.

6.5.2. Dynamically Unresolved Loads and Stores

The ARB supports dynamically unresolved loads and stores in processors that have provision for recovery (which may have been provided to support speculative execution of code or for fault-tolerant purposes). Thus, the ARB allows a load to be executed the moment its address is known, and even before the addresses of its preceding stores are known. Similarly, it allows a store to be executed the moment its address and store value are both known. Allowing dynamically unresolved loads is especially important, because loads often reside in the critical path of programs, and undue detainment of loads would inhibit parallelism.

6.5.3. Memory Renaming

The ARB allows the renaming of memory. Because it has the provision to store up to n values per address entry (where n is the number of ARB stages), the processor can have up to n dynamic names for a memory location. Memory renaming is analogous to register renaming; providing more physical storage allows more parallelism to be exploited [18]. The advantage of memory renaming is especially felt in the renaming of the stack. Consider the piece of assembly code in Figure 6.6, in which a parameter for a procedure is passed through the stack. Both instantiations of PROCEDURE1 use the same stack location for passing the parameter. Memory renaming with the help of ARB allows these two instantiations to be executed at the same time or even out-of-order; if memory renaming were not possible, then the two instantiations become serialized. The full potential offered by the memory renaming capability of the ARB is a research area that needs further investigation.

```
             STORE  R1, 0(SP);     Place parameter in stack
             CALL   PROCEDURE1
             ...
             STORE  R2, 0(SP);     Place parameter in stack
             CALL   PROCEDURE1
             ...
PROCEDURE1:  LOAD   R3, 0(SP);     Retrieve parameter from stack
             ...
```

Figure 6.6: Assembly Code for Depicting the Benefits of Memory Renaming

6.6. ARB Extensions

6.6.1. Handling Variable Data Sizes

Many instruction set architectures allow memory references to have byte addressability and variable data sizes, such as bytes, half-words (16 bits), and full-words (32 bits). If a machine supports partial-word stores, and the ARB keeps information on a full word-basis (it is easier for the ARB to keep information on a full word-basis if most of the memory references are to full words), then special care has to be taken in the ARB. When a partial-word store is executed, the store value stored in the ARB should be the full-word value and not the partial-word value. This is because subsequent loads to the same word may require the full-word value. The only additional support needed to support partial-word stores is that the ARB should treat a partial-word store as the combination of a full-word load followed by a full-word store. Thus, when a partial-word store with sequence number i is executed, both the load bit and the store bit of stage i in the ARB entry are set to 1.

6.6.2. Set-Associative ARBs

The discussion so far was based on fully-associative ARB banks (notice that although each ARB bank is fully associative, the overall ARB is functionally set-associative because of the interleaving). It is possible to implement set-associative ARB banks much like set-associative caches. Figure 6.7 shows the block diagram of a two-way set-associative ARB. In the figure, each ARB bank contains two sets, each of which contains two rows. When a memory reference is sent to the ARB, the proper ARB bank and the set are determined, and the associative search for the address is performed only within the identified set in the identified bank. If the identified bank is full, then special recovery actions are taken as described in Section 6.4.4. Set-associative ARBs help to reduce the complexity of the associative search for addresses in the ARB. The ARB banks can even be made direct-mapped so as to facilitate an even faster lookup for memory references. However, a direct-mapped ARB may initiate recovery actions more often because of many addresses getting mapped to

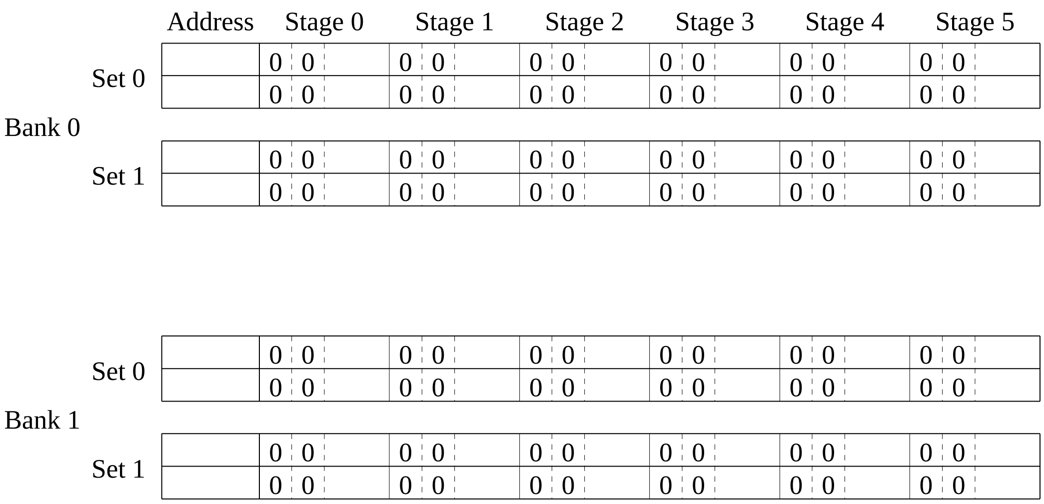


Figure 6.7: A Two-Way Interleaved, Two-Way Set Associative, 6-stage ARB

the same ARB row entry.

Increasing the degree of interleaving also helps to reduce the width of the associative search, besides allowing more memory references to be issued per cycle. It is worthwhile to carry out performance studies to study the impacts due to different ARB sizes (number of sets), interleaving degrees, set associativities, and stages on different execution models; we leave this as a subject for future research.

6.7. Summary

A high-bandwidth data memory system is crucial to the performance of the multiscalar processor. The data memory bandwidth can suffer greatly if the top-level data cache is a standard blocking cache because of its serial services of misses. To reduce this bandwidth degradation, we considered non-blocking caches. To further improve the bandwidth of the data memory system to more than one request per cycle, we interleaved the data cache to create a multi-ported cache. This multi-port, non-blocking data cache allows multiple memory requests to be serviced in a single cycle, irrespective of whether they are hits or misses.

We have proposed a hardware structure called an Address Resolution Buffer (ARB) for disambiguating memory references at run time. The ARB scheme has the full power of a hypothetical scheme that allows out-of-order loads and stores by performing associative compares of the memory addresses of all loads and stores in the entire instruction window. It allows speculative loads and speculative stores! Furthermore, it allows memory references to be executed before they are disambiguated from the preceding stores and even before the addresses of the preceding stores are determined. It also allows forwarding of memory values when loads are performed, and all stores are effectively ‘‘cache hits’’ because the values are stored only in the ARB (‘‘cache besides the cache’’) until they are committed. The ARB also allows memory renaming; it allows as many names for a memory location as the number of stages in the ARB. The ARB is, to the best of our knowledge, the first decentralized design for performing dynamic disambiguation within a large window of instructions.

The ARB mechanism is very general, and can be used in a wide variety of execution models. The two-level hierarchical ARB used in the multiscalar processor uses the concept of splitting a large task (memory disambiguation within a large window) into smaller subtasks, and it ties well with the multiscalar execution model. We also proposed several extensions to the ARB, such as set-associative ARBs and extensions for handling variable data sizes.

Chapter 7. Empirical Results

How well does the multiscalar processor perform?

The previous chapters described the multiscalar paradigm and a multiscalar processor. Perhaps this is the place to start assessing the potential of the multiscalar processor in more concrete terms. A sound evaluation of the multiscalar execution model can be done only after the development of a compiler that considers the idiosyncrasies of the model and exploits its potential. The development of the compiler (by other members of the multiscalar research team) is in the beginning stages, and therefore the results we present in this chapter should only be viewed as a realistic starting point.

This chapter is organized in five sections. Section 7.1 describes the experimental framework we used for our simulation studies. Section 7.2 presents the performance results obtained for both non-numeric and numeric benchmarks. In particular, we present the instruction completion rate, task prediction accuracies, and the percentage of incorrect loads obtained with different number of execution units, We also present results showing the efficacy of the multi-version register file. Section 7.3 summarizes the chapter.

7.1. Experimental Framework

We shall first discuss the experimental framework used for our studies. Throughout the course of this research, we had developed tools to evaluate the performance of the multiscalar processor and to study the effects of different design modifications. Our methodology of experimentation is simulation, and is illustrated in Figure 7.1. Benchmark programs are compiled to MIPS R2000-R2010 executables using the MIPS compiler. The task generator program takes the MIPS executables, decides the tasks, and demarcates the tasks in the executables. This modified executable is then fed to the

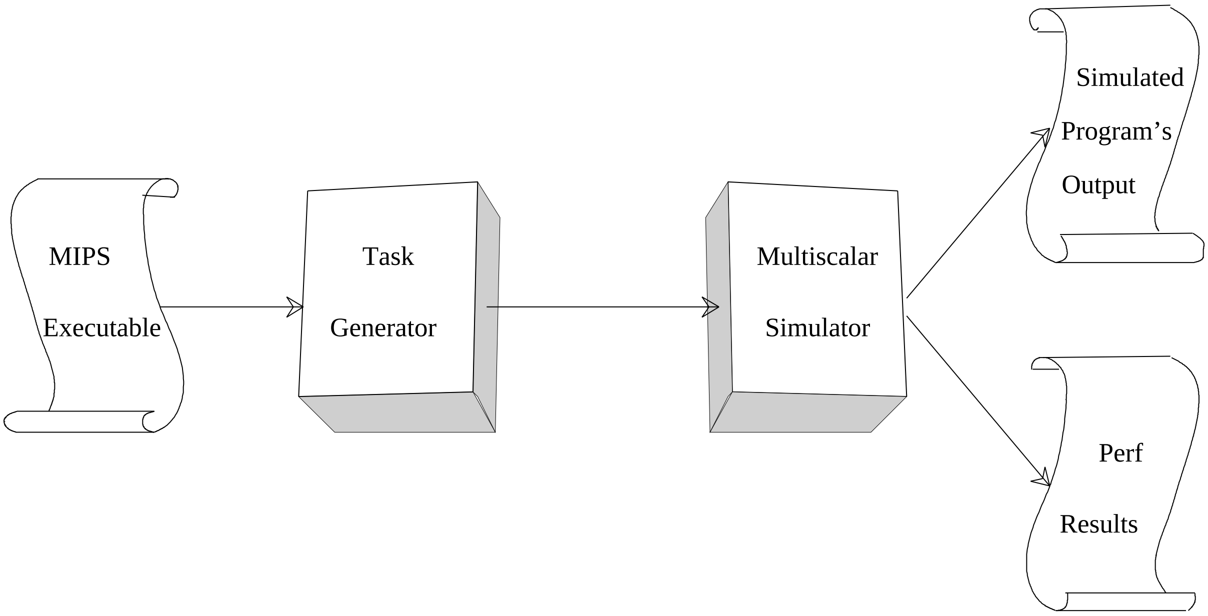


Figure 7.1. Simulation Methodology

multiscalar simulator. The multiscalar simulator is described below in detail.

7.1.1. Multiscalar Simulator

The multiscalar simulator is the heart of the experimental framework. It uses the MIPS R2000 - R2010 instruction set and functional unit latencies [82]. All important features of the multiscalar processor, such as the control unit, the execution units, the distributed instruction caches, the ARB, and the data cache, are included in the simulator. The simulator accepts executable images of programs (with task demarcations), and simulates their execution; it is not trace driven. It is written in C, and consists of approximately 12000 lines of code. The simulator also incorporates a mini-operating system to handle the system calls made by the simulated program. The mini-operating system directly handles some of the system calls, such as those involving signal handling (SYS_sigvec, SYS_sigstack, SYS_sigblock, etc.), those involving control flow changes (SYS_sigreturn, SYS_execve, etc.), and those requesting for more memory (SYS_brk). For instance, when the simulated program requests for more heap memory, the simulator simply changes the simulated heap pointer. On the other hand, the remaining system calls, such as those for handling file I/O (SYS_open, SYS_close, SYS_read, SYS_write) are passed on to the main operating system with a modified set of parameters. In any case, the code executed during the system calls are not taken into account in the performance results. The simulator can run in interactive mode or in batch mode. In interactive mode, a debugging environment is provided to support single cycle execution, breakpointing, examining registers, etc. In batch mode, the simulator collects performance data. The simulator is able to accurately account for the time to do instruction fetching, decoding, and execution. Furthermore, the simulator is parameter-driven; most of the parameters can be specified through command line arguments, and the rest through parameter files. The important parameters, which can be varied, are listed below:

- Number of execution units
- Number of instructions fetched and decoded per cycle in an execution unit
- Number of instructions issued per cycle in an execution unit
- In-order execution or out-of-order execution within an execution unit.
- Maximum static size of a task
- Instruction cache size, access latency, and miss latency
- Data cache size, access latency, and miss latency

7.1.2. Assumptions and Parameters Used for Simulation Studies

The simulations take a long, long time to run (approximately 2 days to simulate 100 million instructions). Therefore, we had to drastically restrict the design space explored. We also chose and fixed some parameters such that they do not distract us from the tradeoffs under investigation. For this study, we decided not to investigate the effects of cache size, cache associativity, and task size. The parameters that were fixed for this study are listed below:

- The benchmarks are run for 100 million instructions each (less for *compress*, as it finishes earlier;

500 million for *spice2g6*, as its results do not stabilize until then)

- A task can have up to 32 instructions
- Out-of-order execution is performed in each unit
- Each execution unit has a 4Kword local instruction cache, with a miss latency of **4 cycles**
- 100% hit ratio is assumed for the global instruction cache
- The common, top-level data cache is 64Kbytes, direct-mapped, and has an access latency of **2 cycles** (one cycle to pass through the interconnect between the execution units and the ARB, and another to access the ARB and the cache in parallel). The interleaving factor of the data cache and the ARB is the smallest power of 2 that is equal to or greater than twice the number of execution units. For instance, if the number of units is 12, the interleaving factor used is 32.

7.1.3. Benchmarks and Performance Metrics

For benchmarks, we used the SPEC '92 benchmark suite [104] and some other programs, namely *Tycho* (a sophisticated cache simulator) [69] and *Sunbench* (an Object Oriented Database benchmark written in C++) [26]. The benchmarks and their features are listed in Table 7.1. Although our emphasis is on non-numeric programs, we have included some numeric benchmark programs for the sake of comparison. The benchmark programs were compiled for a DECstation 3100 using the MIPS C and FORTRAN compilers. Notice that the **`a.out`** executables so obtained have been compiled for a single-issue processor.

For measuring performance, *execution time* is the sole metric that can accurately measure the performance of an integrated software-hardware computer system. Metrics such as instruction issue rate and instruction completion rate, are not very accurate in general because the compiler may have introduced many redundant operations. However, to use execution time as the metric, the programs have to be run to completion, which further taxes our already time-consuming simulations. Furthermore, we use the same executable for all the simulations, with no changes instituted by a compiler. Due to these reasons, we use instruction completion rate as the primary metric, with the following

Table 7.1: Benchmark Data

Benchmark	Input File	Language	Type
compress	in	C	Non-numeric
eqntott	int_pri_3.eqn	C	Non-numeric
espresso	bca	C	Non-numeric
gcc	stmt.i	C	Non-numeric
sc	loada1	C	Non-numeric
sunbench	sunbench.output.og	C++	Non-numeric
tycho	trace of ‘xlisp li-input.lsp’	C	Non-numeric
xlisp	li-input.lsp	C	Non-numeric
dnasa7		FORTRAN	Numeric
doduc	doducin	FORTRAN	Numeric
fpppp	natoms	FORTRAN	Numeric
matrix300		FORTRAN	Numeric
spice2g6	greycode.in	FORTRAN	Numeric

steps adopted to make the metric as reliable as possible: (i) nops are not counted while calculating the instruction completion rate, (ii) speculatively executed instructions whose execution was not required are not counted while calculating the instruction completion rate. Thus, our main metric can be called as *useful instruction completion rate (UICR)*.

We also use three other metrics — task prediction accuracy, percentage useless work, and percentage incorrect loads — to throw more light on the multiscalar processor.

7.1.4. Caveats

It is important to make note of some of the caveats in our performance study. The performance of any computer system is highly dependent on the compiler. We have, however, not used a multiscalar compiler for our performance studies; as the compiler is still being developed by other members of the multiscalar research team. We had only two options while writing this thesis — either do no performance studies or do performance studies with the existing compiler (for a single-issue processor). And, we opted for the second choice. Thus, all our studies are carried out with code compiled for a single-issue processor (MIPS R2000). The implications of this decision are: (i) the task selection does not use any intelligence other than the control flow structure of the program, (ii) no multiscalar-specific optimizations were performed on the code, and (iii) the code is scheduled for a single-issue processor, which can have serious implications on multiscalar performance, as illustrated in the next chapter where we discuss the role of the compiler. Further, the tasks used for this simulation study do not have loops embedded within them. Therefore, the performance results presented here should only be viewed as a realistic starting point, and the results should be interpreted with caution. With the description of the experimental setup in place, we can begin the more exciting task of analyzing the results.

7.2. Performance Results

In this section, we present different performance results obtained with our simulation experiments. As there are too many different parameters that can be varied, it is not possible to run experiments with all possible parameter changes. Nevertheless, we have attempted to cover an interesting range of the spectrum.

7.2.1. UICR with One Execution Unit

As a starting point, we shall first present the useful instruction completion rates (UICR) obtained with a single execution unit. Figure 7.2 presents these results. The X-axis shows the different benchmarks, and the Y-axis denotes the useful instruction completion rate. The black shade indicates the useful completion rates obtained by issuing at most one instruction per cycle in the execution unit, and the light shade indicates the completion rates obtained by issuing up to two instructions per cycle in the execution unit. The data cache miss latency to get the first word is 4 cycles (assuming the presence of a second level data cache). These results can be used as a base for comparing results (the completion rates with the single-issue execution unit should be slightly less than the completion rates of an ordinary single-issue processor, as the multiscalar processor has some overheads in task allocation and deallocation).

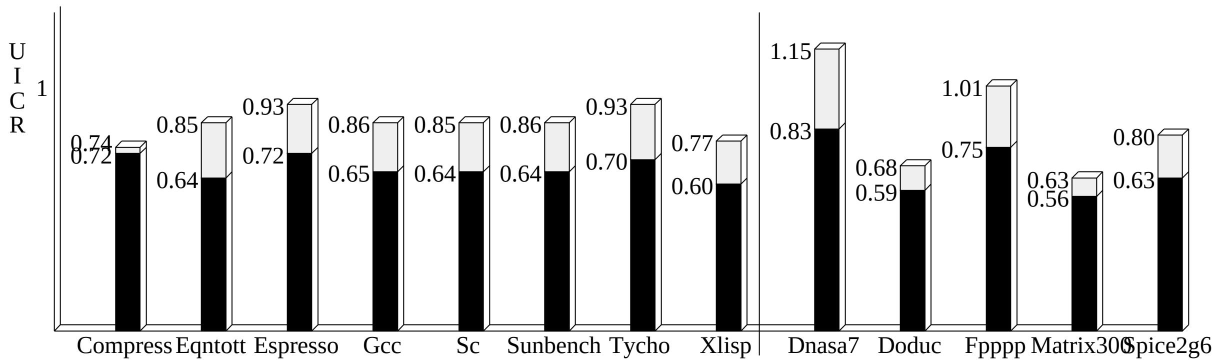


Figure 7.2: Useful Instruction Completion Rates with One Execution Unit

7.2.2. UICR with Multiple Execution Units

In this section, we consider the results obtained by varying the number of execution units. Figure 7.3 presents the useful instruction completion rates obtained for the benchmark programs for different number of execution units. The different shades indicate the useful completion rate obtained with 1, 2, 4, 8, and 12 execution units, each of which can fetch and issue up to 2 instructions per cycle. The darkest shade corresponds to 1 execution unit, the next darkest corresponds to 2 execution units, and so on. For example, the useful instruction completion rate with 1, 2, 4, 8, and 12 execution units for *espresso* are 0.93, 1.44, 1.96, 2.98, and 3.74, respectively. For all these experiments, the cache miss latency is again 4 cycles.

Let us look at the results in some detail. The results for the non-numeric programs are impressive, considering that no multiscalar-specific optimizations were made and that the simulator takes into account all important aspects of the multiscalar processor, and considers the actual functional unit latencies of MIPS R2000-R2010. The actual speedup compared to a single-issue processor with

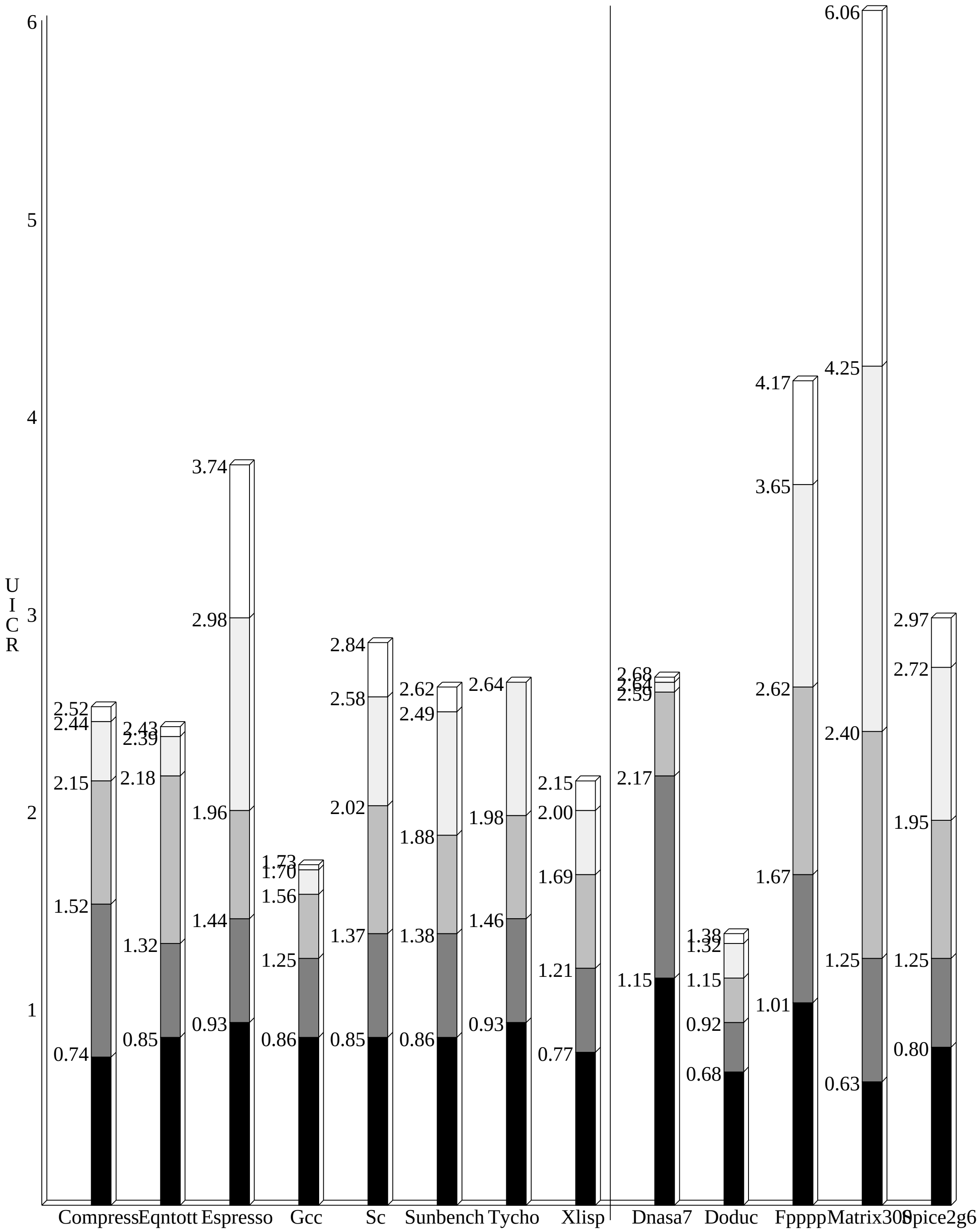


Figure 7.3: Useful Instruction Completion Rates with Multiple Execution Units

the same cycle time will be higher than the useful instruction completion rates presented here. It is important to note that espresso and eqntott show linear speedup improvements, whereas most of the rest seem to saturate around 12 execution units, *for code compiled for a single-issue machine and with no multiscalar-specific optimizations.*

Although the multiscalar processor was designed primarily with non-numeric programs in mind, it is worthwhile to see how it performs for numeric programs. As we can see from the right hand portion of Figure 7.3, the results are not very impressive (except for matrix300), considering that numeric programs generally contain regular parallelism, which is easy to exploit. Why did the multiscalar processor perform poorly for the numeric benchmarks? The reason is that the executable we used for our simulations is scheduled for a single-issue machine. The numeric benchmarks contain many loops in which a loop induction variable is used to access arrays and perform various operations on the array elements. The MIPS compiler typically places the instruction updating the loop induction variable at the end of a loop body. Because the multiscalar task generator invariably considers each iteration to be a separate task, the loop induction variable update happens at the end of the task. Subsequent tasks have to wait for this update to happen, and thus most of the execution in the multiscalar processor gets serialized. The results presented in chapter 8 with rescheduled code bear testimony to this fact.

It is important to note that the instruction completion rates reported here were obtained with the actual floating point functional unit latencies of the MIPS R2010 co-processor. Thus, the actual speedups, compared to a single-issue MIPS processor, both having the same cycle time, would be substantially higher than the completion rates reported here.

7.2.3. Task Prediction Accuracy

Next, we shall consider the task prediction accuracy. Table 7.2 presents the task prediction accuracies obtained with different number of execution units. The task prediction accuracy is measured as follows: when a task T is committed in execution unit *i*, a check is made to determine if T was

Table 7.2: Task Prediction Accuracies and Percentage Useless Instructions Executed

Benchmarks	Task Prediction Accuracy (%) With Number of Execution Units =					Useless Instructions Executed (%) With Number of Execution Units =				
	1	2	4	8	12	1	2	4	8	12
compress	88.90	88.90	88.91	88.91	88.85	0.00	5.78	18.16	18.61	47.47
eqntott	97.50	97.51	97.50	97.50	97.50	0.00	2.50	7.04	15.01	20.35
espresso	96.78	96.74	96.80	96.84	96.89	0.00	1.53	4.96	9.96	15.91
gcc	83.88	83.63	83.80	83.81	83.99	0.00	7.91	23.38	38.03	43.64
sc	96.23	95.90	95.68	96.26	97.10	0.00	2.04	6.22	14.56	19.66
sunbench	94.28	94.98	94.81	94.90	95.11	0.00	3.54	10.42	22.17	31.75
tycho	98.53	98.48	98.41	98.40	98.41	0.00	2.82	8.10	17.26	25.21
xlisp	93.38	93.36	93.35	93.37	93.31	0.00	4.35	12.65	27.01	37.82
dnasa7	98.36	98.36	98.36	98.36	98.36	0.00	0.08	0.27	2.01	2.87
doduc	80.33	80.23	81.62	81.68	81.90	0.00	4.25	13.93	26.31	26.82
fpppp	98.71	98.82	98.77	98.33	98.81	0.00	1.36	5.88	16.78	27.38
matrix300	99.98	99.66	99.66	99.66	99.66	0.00	0.00	0.05	0.25	0.25
spice2g6	99.01	98.99	98.99	98.99	99.00	0.00	0.61	2.38	5.52	7.26

the first task allocated to unit *i* after the previous task in *i* was committed. Thus, the task prediction accuracy is measured only for the committed tasks. Although the task prediction accuracy gives some indication of the fraction of time useful work is done, this metric by itself does not give much information as far as as the multiscalar processor is concerned. This is because, rather than knowing that a misprediction occurred, it is more important to know where in the circular queue the misprediction occurred (or better still is to know how much work was wasted because of the misprediction). Therefore, we also show in Table 7.2 the amount of useless instructions executed, *i.e.*, the number of instructions that were executed and later squashed (because of incorrect task prediction) as a percentage of the total number of instructions executed.

The results presented in Table 7.2 throw further light on why the performance of the non-numeric benchmarks dropped when the number of execution units was increased. We can see that the prime cause for the drop in instruction completion rates is the increase in useless work done because of incorrect task prediction. For instance, when 12 execution units were used for *gcc*, 43.64% of the work done was useless (with no multiscalar-specific optimizations done). Although the task prediction accuracy remains more or less constant irrespective of the number of execution units, by the time several predictions are made in a row, the probability that all the predictions are correct diminishes. Thus, the task prediction accuracy has to be improved if more performance is to be obtained by the use of additional execution units.

7.2.4. Incorrect Loads

Recall from chapter 6 that in the multiscalar processor, the loads are executed before their addresses are disambiguated from the store addresses in the preceding tasks. The ARB is used to detect any out-of-sequence load-store access to the same memory location, and initiate appropriate recovery actions. It is worthwhile to see how many of these unresolved loads fetched incorrect values. Table 7.3 presents the percentage of loads that were incorrect, and caused the ARB to initiate recovery actions, for different number of execution units. From the results we can see that the percentage of incorrect loads is low when the number of execution units is less than or equal to 4, and somewhat higher when the number of execution units is more than 4. When 12 execution units are used, the percentage of incorrect loads varies from 0.07% to 10.46%. If the loads were executed after disambiguating them from all preceding stores, then it would almost have amounted to serial execution in the multiscalar processor. Thus there is merit in executing the loads as dynamically unresolved loads using the ARB.

7.2.5. Effect of Data Cache Miss Latency

In order to study how sensitive the multiscalar processor is to the first level cache miss latency, we conducted another set of experiments with 4 execution units for miss latencies of 8 cycles and 12 cycles (for the first level cache). The remaining parameters are same as that in Figure 7.2. Figure 7.4 presents the useful instruction completion rates so obtained. The results obtained with a miss latency

Table 7.3: Out-of-Sequence Loads that are Incorrect

Benchmarks	Percentage of Loads that are Incorrect				
	1	2	4	8	12
compress	0.00	0.00	0.58	5.16	6.19
eqntott	0.00	0.00	0.14	0.28	0.27
espresso	0.00	0.02	0.61	1.89	3.04
gcc	0.00	1.68	4.13	6.14	6.64
sc	0.00	1.55	2.53	5.89	8.70
sunbench	0.00	0.17	0.93	6.68	10.46
tycho	0.00	0.75	3.81	7.67	8.18
xlisp	0.00	1.12	1.70	3.70	5.09
dnasa7	0.00	0.17	0.23	0.46	0.52
doduc	0.00	0.72	2.21	2.67	3.09
fpppp	0.00	1.24	2.82	5.10	7.43
matrix300	0.00	0.00	0.00	0.00	0.07
spice2g6	0.00	0.32	2.59	2.76	3.03

of 4 cycles are also presented for comparison purposes. The different shades indicate the useful completion rate obtained with 12, 8, and 4 cycle miss latencies. The darkest shade corresponds to 12 cycles, the next darkest corresponds to 8 cycles, and so on. The performance results seem to be almost unaffected by the miss latency of the first level data cache. It is important to remember, however, that the code has been compiled for a single-issue processor, and it is quite possible that the performance is constrained for all the cases because the scheduling has been done with a single-issue processor in mind.

7.2.6. Effect of Data Cache Access Time

In order to study how sensitive the multiscalar processor is to the first level cache access time, we conducted another set of experiments with 4 execution units for an access time of 4 cycles for the first level cache. The remaining parameters are same as that in Figure 7.2, except that the cache miss latency was 8 cycles. Figure 7.5 presents the useful instruction completion rates so obtained. The

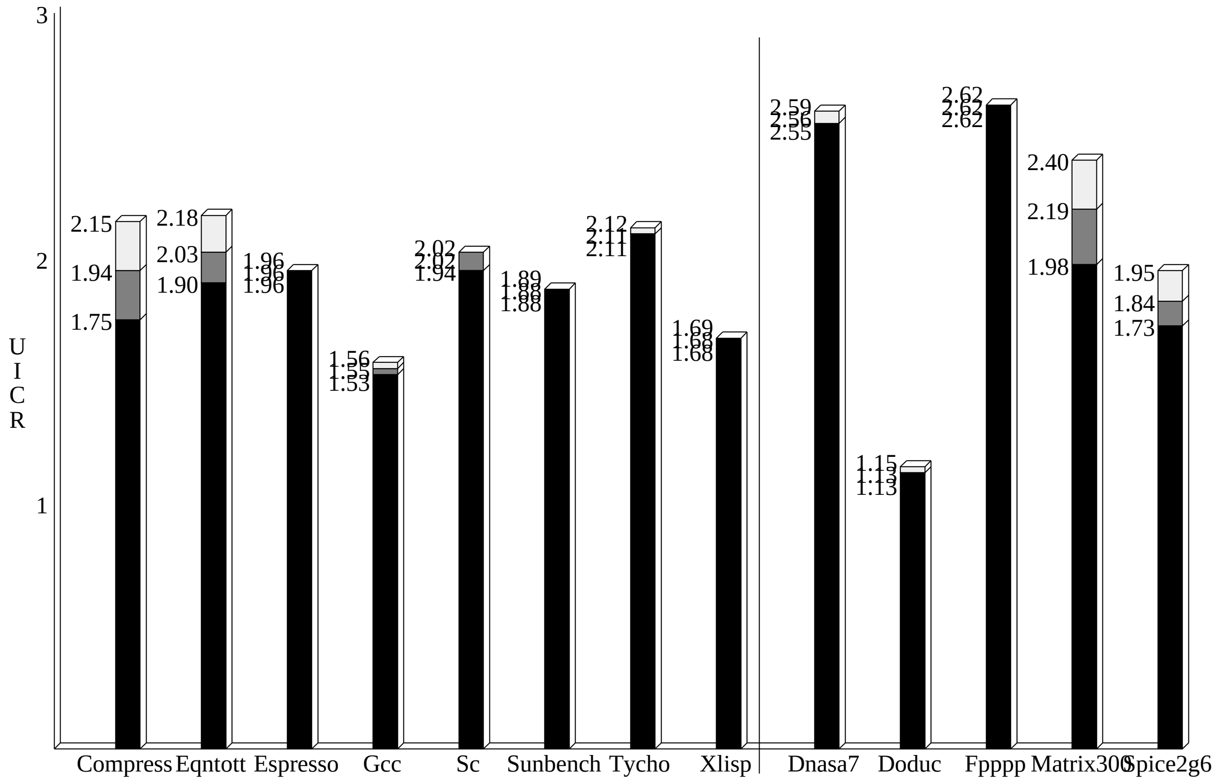


Figure 7.4: Useful Instruction Completion Rates with Different Miss Latencies

results obtained with an access time of 2 cycles are also presented for comparison purposes. The dark shade corresponds to results obtained with an access time of 4 cycles, and the light corresponds to the results with an access time of 2 cycles. The performance results are affected by as low as 0% (for *dnasa7*) to as high as 22% (for *xlisp*) when the access time of the first level data cache is changed from 2 cycles to 4 cycles. Again, it has to be taken into consideration that the code has been compiled for a single-issue processor, with no multiscalar-specific optimizations applied.

7.2.7. Efficacy of Multi-Version Register File

The final set of results deals with establishing the efficacy of the multi-version register file. The central idea behind the multi-version register file was to exploit the communication localities present

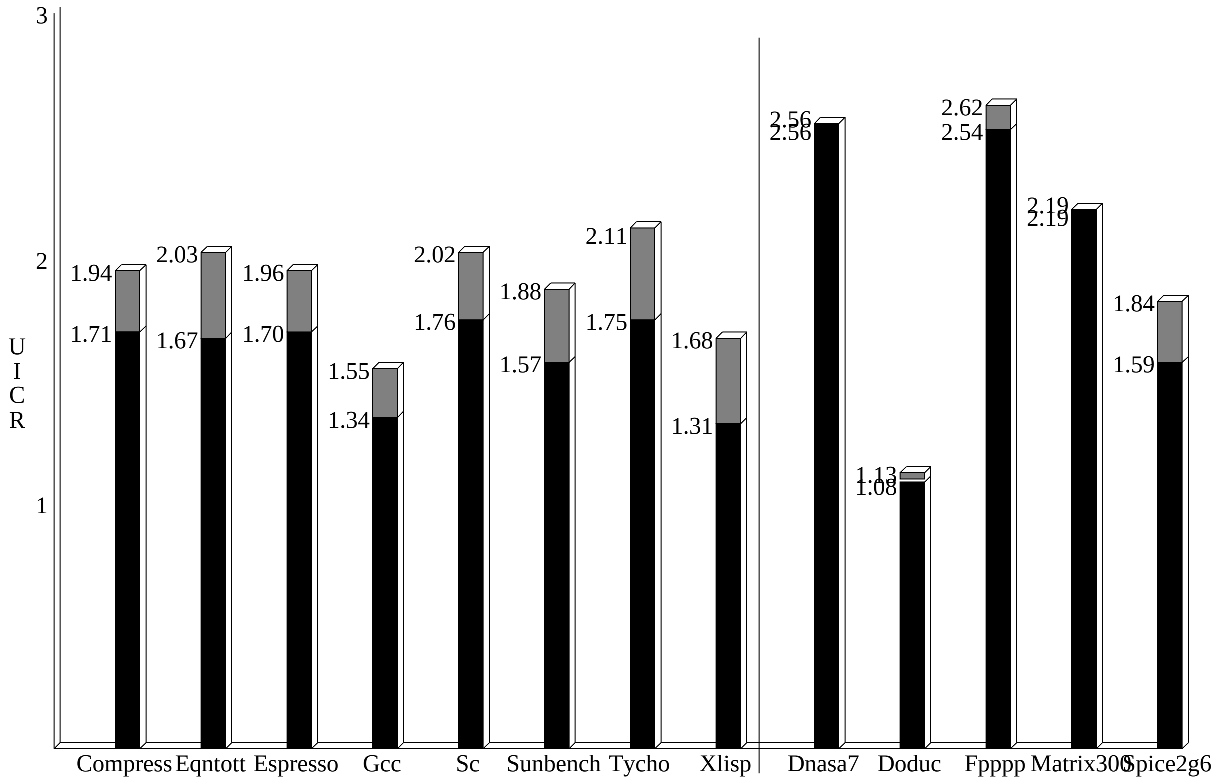


Figure 7.5: Useful Instruction Completion Rates with Different Cache Access Times

in the programs to decentralize the register file. Instead of potentially slowing down all register accesses by means of a centralized register file, the multi-version register file provides fast access to ‘‘local’’ data, and increasingly slower access to ‘‘remote’’ register data.

In order to study the efficacy of the multi-version register file, we conducted another set of experiments with 4 execution units and 8 execution units, assuming a hypothetical, centralized register file that has the capability to store all the versions required to support multiscalar execution. This register file is assumed to have as many ports as required, and the same access time as that of a register access in the multi-version register file scheme. The remaining parameters are same as that in Figure 7.2. Figure 7.6 presents the useful instruction completion rates obtained with such a register file. The results obtained with the multi-version register file are also presented for comparison purposes.

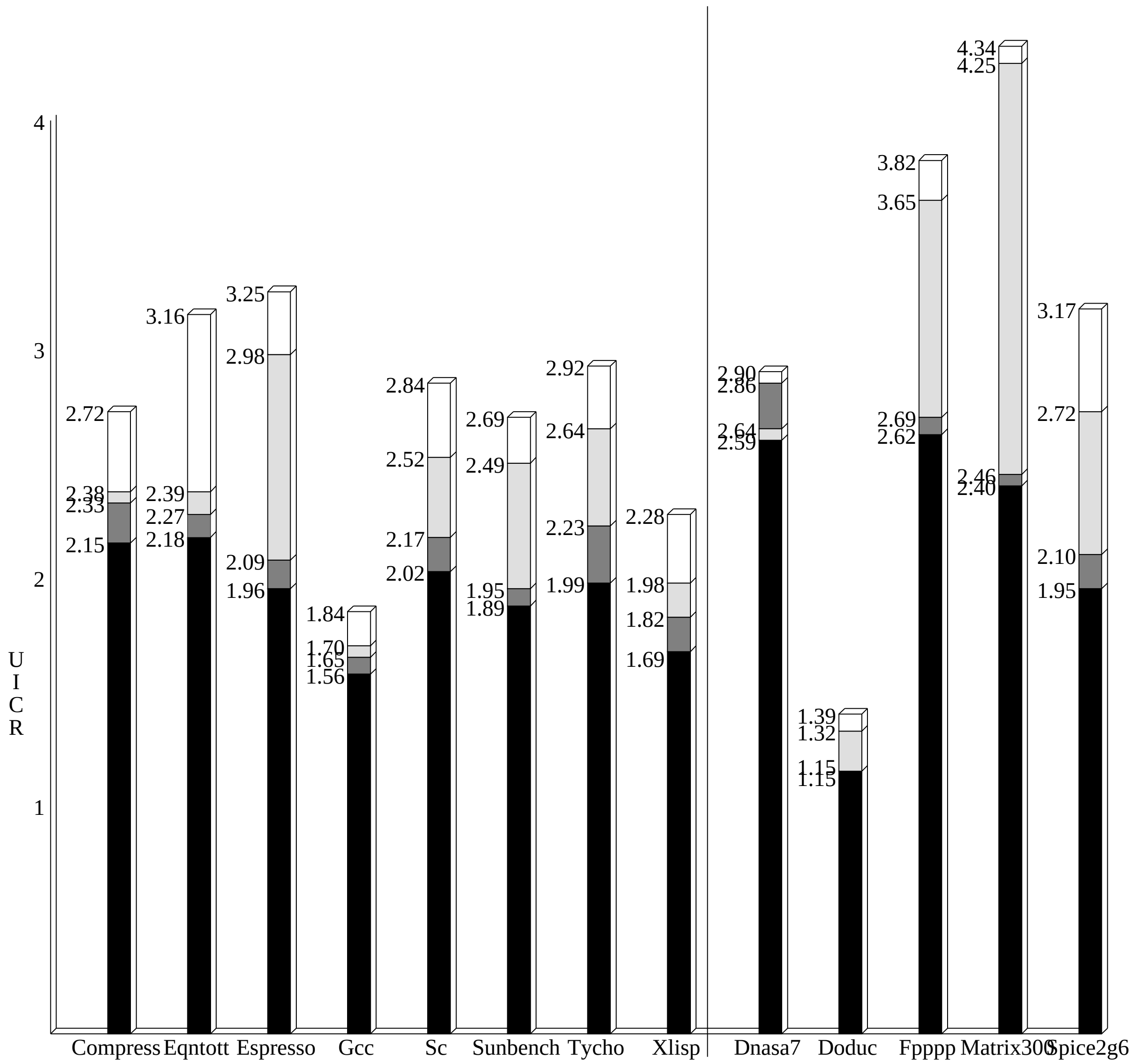


Figure 7.6: Useful Instruction Completion Rates with a Multi-Version RF and a Centralized RF

In the figure, the dark shade corresponds to the results with the multi-version register file and 4 execution units; the next dark shade corresponds to the results with a centralized register file (having the same access time) and 4 execution units. The next dark shade corresponds to the results with a multi-version register file and 8 execution units, and the unshaded region corresponds to the results with a centralized register file and 8 execution units. From the figure, we can see that with 4 execution units, for some programs there is little or negligible performance penalty (*eqntott*, *doduc*, *fpppp*, and *matrix300*) due to the multi-version register file, and for the rest of the programs, there is at most a 12% performance penalty (for *tycho*) *using code compiled for a single-issue processor*. With 8 execution units, however, the performance difference is more pronounced. Our experience with the multiscalar simulations leads us to believe that this difference can be significantly reduced if each execution unit is allowed to access the local register file of the previous unit in addition to accessing its own local register file. The performance penalty would decrease further if the compiler takes into consideration the inter-task register dependencies when generating the tasks.

It is worthwhile to refer to some results that we had published earlier [59], showing the efficacy of the multi-version register file. It was shown that a large fraction of the operands were generated in the same task or came from at most 3 tasks in the past. Similarly, a large portion of the register instances were used up and overwritten either in the same unit in which they were created or in the subsequent unit. Thus most of the register instances need not be even forwarded to the subsequent units, and among the forwarded ones, most were forwarded to at most one execution unit. Thus, a significant portion of the register read traffic falls under intra-task communication, indicating that the multi-version register file system does not present a bottleneck to performance. These results suggest that the multi-version register file provides sufficient bandwidth to meet the demands of the multiscalar processor, demands that would have been difficult to meet with a centralized register file.

7.4. Summary

This chapter presented the performance results for the multiscalar processor, obtained through a simulation study. The performance of any computer system is sensitive to the compiler and the optimizations it performs. As a compiler for the multiscalar processor is still in development, we used for the simulation study program executables that were compiled for the DECstation 3100, a single-issue machine. Further, no multiscalar-specific optimizations were done for the same reason. The next chapter gives a feel of the performance improvements that could be obtained if multiscalar-specific optimizations are performed.

First we presented the useful instruction completion rate obtained with a single execution unit. Then we presented different performance results obtained with 1, 2, 4, 8, and 12 execution units. The results were generally impressive, considering that no multiscalar-specific optimizations were done. Then we presented some performance results to show how much the performance is sensitive to the data cache miss latency. These results suggest that the performance is not very sensitive to the data cache miss latency, for code scheduled for a single-issue processor. Finally, we presented some results to study the efficacy of the multi-version register file. The results show that the performance with the multi-version register file is comparable to the performance with a hypothetical, centralized register file that has the same register access time.

Chapter 8. Role of Compiler

How can the compiler improve performance?

The multiscalar paradigm is grounded on a good interplay between compile-time extraction of ILP and run-time extraction of ILP. This was best brought out by the illustration in Figure 3.6, which showed what was done by the compiler and what was done by the hardware. The compiler has a big role to play in bringing to reality the full potential of the multiscalar paradigm. The role of the compiler in the multiscalar paradigm is to: (i) generate tasks, of suitable size, to facilitate the construction of a large and accurate dynamic window and to minimize inter-unit dependencies, (ii) schedule each task so that run-time stalls due to inter-unit dependencies are minimized, and (iii) perform other support work such as task prediction as creation of register create masks.

8.1. Task Selection

The performance of the multiscalar processor is highly dependent on how the CFG of a program is divided into tasks. Task selection involves packing nodes from the CFG of a program into a task. The definition of a task being an arbitrary connected subgraph of the CFG, the compiler has a wide choice to work with. The tasks could be sequences of basic blocks, entire loops, or even entire function calls. Several factors need to be considered in selecting the tasks, the major ones being: (i) reasonable dynamic task size, (ii) minimum variance for the dynamic task size, (iii) minimum inter-task data dependencies, (iv) minimum inter-task control dependencies so as to allow the the multiscalar hardware to follow independent flows of control. These criteria are further explained below. In order to better satisfy these criteria, the compiler can modify the CFG by performing *global scheduling*.

8.1.1. Task Size

The size of a task is a crucial issue; small task sizes will not be able to sustain significant amounts of ILP because the sustained ILP is upper bounded by the average dynamic task size. An optimum value for the average execution time of a task is a value that is approximately equal to the number of execution units in the multiscalar processor. If the average execution time is significantly less than the number of units, then by the time the global control unit assigns one round of tasks, the units that were initially assigned would have finished long time back, and would have been idle for a long time. On the other hand, if the average execution time is significantly greater than the number of units, then by the time the global control unit finishes one round of task assignments, it has to wait for a long time to assign further tasks. Although the units are not idle, and are instead executing the big tasks, big tasks tend to have more inter-task data dependencies. Similarly, variance of the dynamic task size is also a crucial issue. This is because, work is allocated to the execution units in sequential order; if a big task is allocated to an execution unit E_0 followed by execution units E_1 - E_{n-1} with small tasks, then units E_1 - E_{n-1} have to wait until the execution of the big task in E_0 is over, although they had finished the execution of their tasks long time back.

8.1.2. Inter-Task Data Dependencies

The next issue to be considered by the compiler in selecting tasks is inter-task data dependencies, which occur through both registers and memory. The compiler should determine data dependencies and then attempt to minimize inter-task data dependencies, possibly by attempting to pack dependent instructions into a task as much as possible. This will allow multiple tasks to be executed mostly independent of each other, thereby reducing the run-time stalls, and improving the performance of the processor. This also reduces the communication that takes place at run time between the execution units. Determining some of the dependencies in the program may be hard, because of indirect memory references through pointer variables.

8.1.3. Task Predictability

Another important issue in deciding the boundaries of a task is the predictability of the subsequent task. If a task *T* has a single target, then the subsequent task is control-independent of *T*. Executing multiple, control-independent tasks in parallel allows the multiscalar processor to execute multiple, independent flows of control, which is needed to exploit significant levels of ILP in non-numeric applications [90]. However, most non-numeric programs have very complex control structures, and constructing reasonable size tasks with single targets may not be possible most of the time. In such situations, the multiscalar compiler can, as far as possible, attempt to demarcate the task *T* at those points where it is fairly sure of the next task to be executed when control leaves *T* (although it may not know the exact path that will be taken through *T* at run time). Such a division of the CFG into tasks will not only allow the overall dynamic window to be accurate, but also facilitate the execution of (mostly) control-independent code in parallel, thereby allowing multiple flows of control. The construction of tasks could be facilitated by profile-guided optimizations as well as by control dependence analyses that identify independent flows of control.

8.2. Intra-Task Static Scheduling

After performing a judicious division of the CFG into tasks, the compiler has a further role to play in improving the performance by way of intra-task scheduling[7]. It can introduce helpful transformations that are tailored to the idiosyncrasies of the multiscalar overall execution model and the execution model within each unit. The primary objective of intra-task scheduling is to *reduce the run-time stall due to inter-task dependencies*. If each execution unit can issue multiple operations per

---

[7] Although we present intra-task scheduling after task selection, ideally, the compiler should consider intra-task scheduling opportunities and their implications even at the time of deciding the tasks. Many multiscalar-specific optimizations can be done before and after task selection.

cycle, then the intra-task scheduler can also attempt to utilize this feature. Notice that it may be possible to do some amount of intra-task scheduling at run time, using a dynamic scheduling execution model within each execution unit; however, the intra-task hardware scheduler is limited in its capabilities because it cannot schedule an instruction until the instruction has been fetched from the local instruction cache.

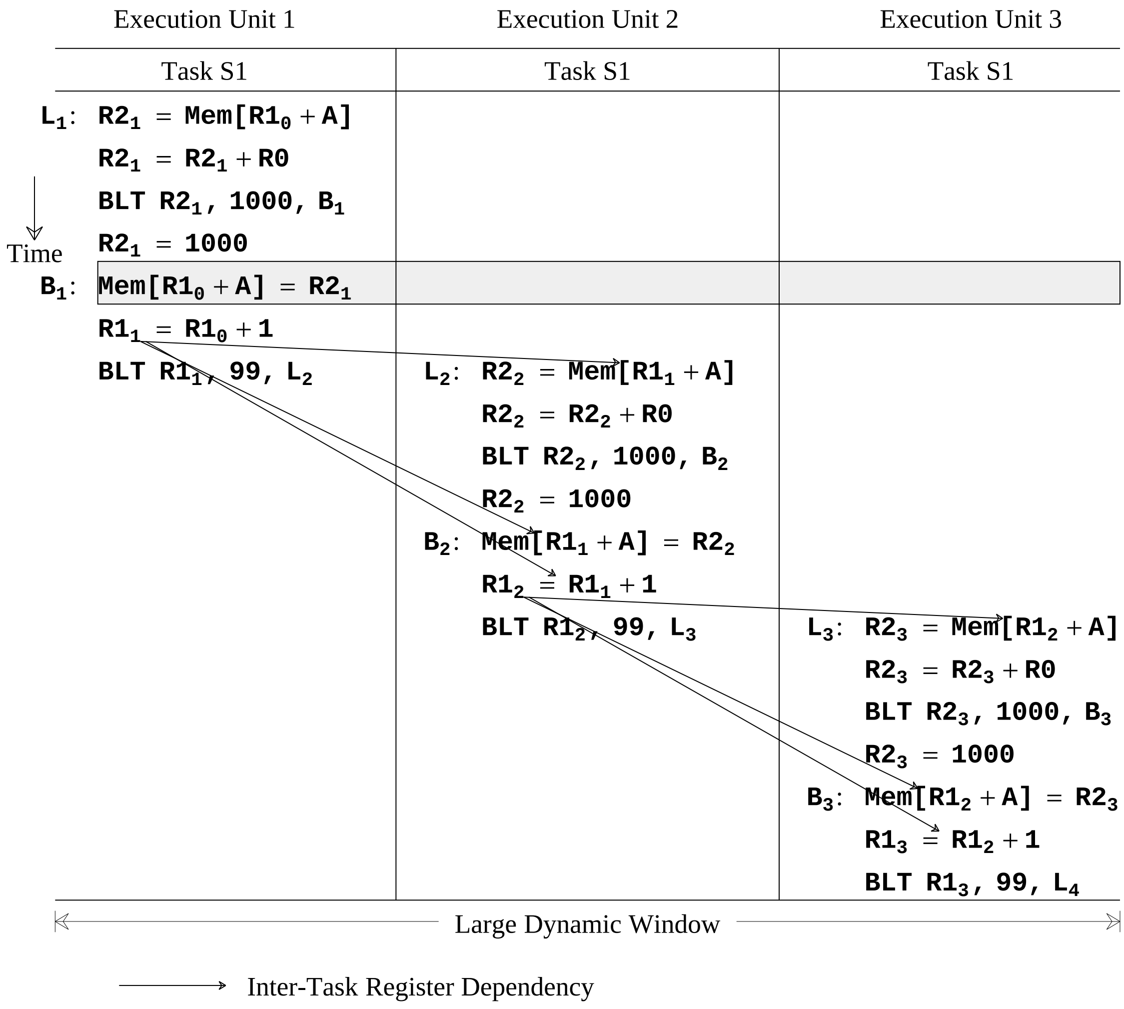
We shall discuss the different considerations involved in intra-task static scheduling. Intra-task scheduling has to consider both inter- and intra-task constraints. Reduction of run-time stalls is achieved by moving up within a task instructions whose results are needed in the beginning of the following tasks, and by moving down within a task instructions whose operands are produced at the end of preceding tasks. Thus, if an instruction I_1 is at the end of a task, and its result is needed by another instruction I_2 in the beginning of the subsequent task, then I_2 will have to wait for a long time. If the execution model within an execution unit is sequential, then the execution of all subsequent instructions are held up, resulting in a sequential overall execution. A prime example where the moving up of operations help is the update of loop induction variables. This is illustrated in the example given below.

Example 8.1: Consider the C code used in Example 3.1. Figure 8.1(i) shows a different assembly code for the same C code, with the update of the loop induction variable (register R1) done more or less at the end of the loop body rather than at the beginning of the loop body. Figure 8.1(ii) shows a multiscalar execution of this assembly code. Compare this with the multiscalar execution in Figure 3.3, and see the effect of updating the loop induction variable at the end of the loop body. The run-time stall and therefore the execution time have drastically increased because of the loop induction update appearing at the end of a task. Therefore, it is imperative to move up such updates to the beginning of the task.

In many cases, moving up or down of instructions is not trivial, and involves doing other transformations. For instance, in the above example, the instruction **`R1 = R1 + 1`** cannot be

```
        R1  =  0           ; initialize induction variable i
        R0  =  b           ; assign loop-invariant b to R0
    L:  R2  =  Mem[R1 + A] ; load A[i]
        R2  =  R2 + R0     ; add b
        BLT  R2, 1000, B   ; branch to B if A[i] < 1000
        R2  =  1000        ; set A[i] to 1000
    B:  Mem[R1 + A]  =  R2 ; store A[i]
        R1  =  R1 + 1      ; increment i
        BLT  R1, 99, L     ; branch to L if i < 99
```

(i) Assembly Code



(ii) Multiscalar Execution

Figure 8.1: Example Code Fragment

moved up within its task by simple reordering, as there are data dependencies to be honored. But, see in Figure 3.2(ii) how it was nevertheless moved up, producing the much better execution of Figure 3.3. The transformation used there was to initialize the loop induction variable to -1 instead of 0. This is just one example of a multiscalar-specific optimization. Many other multiscalar-specific optimizations can be done to aid intra-task scheduling. Although we do not carry out any explicit compiler optimizations or for that matter any compiler work for this thesis, it is useful to learn something of how these optimizations might be done.

8.2.1. Multiscalar-specific Optimizations

The main hurdles in moving up an instruction in a task, especially those involving loop induction updates, are anti-dependencies that occur through the register holding the loop induction variable. In order to facilitate instruction reordering, these anti-dependencies have to be removed using additional registers, if required. Sometimes, this might require the addition of extra instructions to a task. Although the execution of each task then takes extra cycles, the overall execution time may be significantly reduced because the multiscalar processor is better able to overalp the execution of multiple tasks.

Many other compiler optimizations are also possible. If the input operand of an instruction is guaranteed to be present in multiple registers, use as source that register which is likely to reduce run-time stalls and/or allows better schedules. This illustrates the need for multiscalar-specific register uses. Even the criteria for register allocation might be different, and needs further study.

8.2.2. Static Disambiguation

Apart from reordering instructions to reduce run-time stalls due to inter-task register dependencies, the compiler can reorder instructions to reduce the stalls due to inter-task memory dependencies also. Here, the effect of reordering is more pronounced. Consider a load instruction at the beginning of a task *T*. Assume that this load is to the same memory location as a store that is at the end of the preceding task. Ordinary multiscalar execution will most likely execute the load before the store, only to squash later task *T* and the succeeding tasks. All these squashed tasks should be reassigned and re-executed from scratch just because of the incorrect load. Such type of mishaps could be reduced if this dependency information is known at compile time, and the compiler performs appropriate intra-task scheduling of memory references, taking this information into consideration. The compiler can perform static memory disambiguation, and move down (within a task) loads that are guaranteed to conflict with stores in the previous tasks, and move up stores that are guaranteed to conflict with loads in the succeeding tasks. This will be especially useful when the number of execution units is increased.

8.2.3. Performance Results with Hand-Scheduled Code

Table 8.1 gives a feel of the improvements we can expect to see when some of the above points are incorporated. The results in Table 8.1 were obtained by manually performing the optimizations in one or two important routines in some of the benchmarks. Hand code scheduling was performed only for those benchmarks which spend at least 50% of the time in a limited portion of the code, so that manual analysis is possible. The only transformations that was performed was: moving up within a task those instructions whose results are needed in the critical path in the following tasks. (Such instructions typically include induction variables, which are usually incremented at the end of a loop when code is generated by an ordinary compiler.)

Table 8.1: Useful Instruction Completion Rates with Scheduled Code

Benchmarks	Completion Rate with 12 Execution Units	
	Unscheduled Code	Scheduled Code
dnasa7	2.70	9.84
matrix300	6.06	6.42

Even the simple optimizations implemented for Table 8.1 boost performance considerably for *dnasa7*. The completion rate for *dnasa7* has improved from 2.7 to 9.84 with this simple code scheduling. We expect that with such simple compiler enhancements we will be able to double the performance of the C benchmarks also.

8.3. Other Compiler Support

8.3.1. Static Task Prediction

As mentioned in section 4.2.1.1, the compiler can do task prediction, and convey this information to the hardware. If the compiler is able to embed hard-to-predict branches within tasks, then the static task prediction accuracy is likely to be high enough for the multiscalar processor to establish accurate dynamic windows. The compiler can use profile information to make prediction decisions. The big advantage of using static task prediction is that the global control unit can be made very simple, and its functionalities can be even distributed among the local control units to give rise to a fully decentralized control processor.

8.3.2. Reducing Inter-Unit Register Traffic

The compiler can convey to the hardware routine information such as register data dependencies. In particular, it can generate the create mask of a task, and convey it along with the specification of a task. The create mask of a task denotes all the registers for which new values are potentially created in a task.

Apart from conveying register dependency information to the hardware, the compiler can also reduce the register forwarding traffic between the execution units. It can do a live analysis of the register instances produced in each task. If an instance is guaranteed to be dead when control flows out of the task, then that instance need not be forwarded to the subsequent execution units, and the compiler can convey this information to the hardware.

8.3.3. Synchronization of Memory References

In section 8.2.2, we discussed how the compiler could do static disambiguation of memory references for the purpose of doing intra-task scheduling. In many cases, it might not be possible to move up or down a memory reference (due to other constraints), even though a static analysis confirms that it conflicts with another memory reference in a preceding or succeeding task. A method of eliminating the costly recovery actions due to incorrect memory references is to use explicit synchronization between the create and use of unambiguous memory references. One way of incorporating this is for the compiler to mark such loads and stores, and for the hardware to use full/empty bits, possibly in the ARB, to carry out the synchronization. Such techniques merit further investigation.

8.4. Summary

We have expounded the need for a compiler that takes into consideration the idiosyncrasies of the multiscalar processing paradigm. The compiler has a major role to play in bringing to reality the full potential of the multiscalar paradigm. It analyzes the program, and decides which portions should be lumped together as a task. The main criteria in choosing tasks is to facilitate the creation of a large and accurate dynamic window with reasonable size tasks, having minimum inter-task data dependencies,

After generating the tasks, the compiler can do intra-task scheduling with a view to reduce the run-time stalls that occur due to inter-task data dependencies. It can move up within a task instructions whose results are needed in the beginning of the subsequent tasks, and move down instructions whose operands are produced at the end of the preceding tasks. Several multiscalar-specific optimizations can be used to aid this scheduling.

The compiler can also be used for other purposes, such as static task prediction, conveyance of register dependency information to the hardware, and conveyance of memory dependency information to the hardware.

Although the multiscalar compiler has many tasks to perform, it has more flexibility in operation than the compiler for, say, a VLIW processor or a vector processor. The primary reason is that unlike a VLIW compiler or a vectorizing compiler, which typically needs *guarantees* (of independence) to carry out different transformations, a multiscalar compiler does not. Although it is possible to extend the capabilities of a VLIW processor to allow statically unresolved references, such schemes typically have a high overhead [108]. Streamlined hardware mechanisms (the ARB) that dynamically detect memory data dependency violations and recover are an integral part of the multiscalar processor. Therefore, *accurate* static memory disambiguation techniques, which are a major component of a complex compiler, are *useful*, but not *essential* for the multiscalar processor.

Chapter 9. Conclusions

What did we learn?

This final chapter summarizes the thesis succinctly, and describes directions for future research. This thesis introduced and investigated the multiscalar paradigm, a new paradigm for exploiting instruction level parallelism. It also described in great detail a possible implementation of the multiscalar paradigm, called the multiscalar processor.

9.1. Summary

The basic idea of the multiscalar paradigm is to connect several execution units as a circular queue, and traverse the CFG in steps of subgraphs, as opposed to traversing it in steps of instructions or basic blocks. The way this is done is for the compiler to divide the CFG of a program into (possibly overlapping) subgraphs. At run time, the hardware assigns a subgraph (task) to an execution unit, does a prediction as to which subgraph control is most likely to go next, and assigns that subgraph to the next execution unit, and so on. The assigned subgraphs are executed in parallel by the multiple execution units.

The proposed implementation (multiscalar processor) has several novel features. The important aspect is that in spite of these novel aspects, the hardware is realizable and has no centralized resource bottlenecks. It supports speculative execution, executes multiple flows of control, exploits locality of communication, allows arbitrary dynamic code motion (facilitated by efficient memory access violation detection and recovery mechanisms), and does so with hardware that appears to be fairly straightforward to build. Desirable aspects of the hardware include the decentralization of critical resources, absence of wide associative searches, and absence of wide interconnection/data paths. It also allows easy growth path from one generation to other, with a maximum use of hardware and software components. For example, suppose that it is possible to integrate 4 and 8 execution units (along with supporting hardware) in two consecutive generations, respectively. Going from one (hardware) generation to the next could be as simple as replicating the execution units, and increasing the size of the interconnect[8]. Moreover, we expect multiscalar implementations to have fast clocks, comparable to that of contemporary sequential processors. This is because the multiscalar processor is essentially a collection of sequential processors, and there are no centralized critical resources that can adversely impact the clock from one generation to the next. The multiscalar processor appears to have all the features desirable/essential in an ILP processor as we understand them today; *abstract* machines with these capabilities appear to be capable of sustaining ILP in C programs far beyond 10+ IPC [18, 90].

The preliminary performance results are very optimistic in this regard. With no multiscalar-specific optimizations, a multiscalar processor configuration is able to sustain 1.73 to 3.74 useful instructions per cycle for our non-numeric benchmark programs, and 1.38 to 6.06 useful instructions per cycle for our numeric benchmark programs. Currently, it appears that the multiscalar paradigm is an ideal candidate to help us reach our goal of 10+ IPC, and has tremendous potential to be the paradigm of choice for a circa 2000 (and beyond) ILP processor.

9.2. Future Work

This thesis has introduced and described in detail the multiscalar paradigm and the hardware aspects of the multiscalar processor. The work reported here is far from complete. Further research is needed in many related areas, some of which are described below.

---

[8]A bigger interconnect is a necessary evil in any paradigm if more ILP is to be exploited; the only way around it is to reduce the bandwidth demanded of it, for example, by exploiting locality, as discussed in section 9.2.4.

9.2.1. Multiscalar Compiler

The development of a compiler for the multiscalar processor is perhaps the major work in the immediate future. As mentioned in chapter 7, a multiscalar compiler is being developed by other members of the multiscalar research team. This compiler takes into consideration the idiosyncrasies of the multiscalar paradigm. The important issues to be dealt with by the compiler were already described in detail in chapter 8.

9.2.2. Sensitivity Studies

After the development of the multiscalar compiler, several performance studies need to be done. It is important to conduct sensitivity studies by varying different design parameters, such as the nature of tasks, ARB sizes, cache access latencies, and the execution model within each unit.

9.2.3. Evaluation of ARB

Chapter 6 introduced the ARB and showed how it can be used in the multiscalar processor. The appendix shows how the ARB can be used in other processing models. Further work needs to be done in evaluating the ARB to study the performance impacts due to different ARB sizes (number of sets), interleaving degrees, set associativities, and number of stages on different execution models. Memory renaming and the potential offered by the memory renaming capability of the ARB are also research areas that need further investigation. Finally, we would like to extend the ARB so as to enforce different memory consistency models such as weak consistency [41], data-race-free-0 (DRF0) [5], data-race-free-1 (DRF1) [6], processor consistency [65], and sequential consistency [91] in a multiprocessor system.

9.2.4. Multi-Version Data Cache

A major challenge that we face with the ARB connected as described in chapter 6 is the latency of the interconnection network connecting the execution units to the ARB and the data cache. This latency will increase when the number of multiscalar execution units is increased, resulting in an increase in the execution time of all memory references. The ARB complexity, and hence its latency, also increases with the number of execution units, further increasing the execution time for memory references. We feel that reducing the execution latency for memory references is crucial for improving the performance of non-numeric programs.

The solution we propose to reduce the latency is to use a multi-version data cache, much alike the multi-version register file described in chapter 5. The basic idea of the multi-version data cache is to provide each execution unit with a local data cache, so that most of the memory references occurring in an execution unit can be satisfied by its local data cache. If a memory request misses in the local data cache, then the request is forwarded to a *global data cache*, which is common for all the execution units, much like the global instruction cache. However, unlike the case with instruction caches, data caches introduce the cache coherence problem — the multiple versions (of the data cache) have to be kept coherent when an execution unit updates a memory location. Our solution to this problem is to allow the local data caches to be temporarily incoherent, with provision to update them (and make them coherent) using the forwarding network. Provision is also made to detect situations when accesses are made to incoherent data. When an incorrect access is detected, the processor can use the recovery mechanism provided for incorrect task prediction to initiate corrective action.

Figure 9.1 shows the block diagram of an 8-unit multiscalar processor with the multi-version data cache. Notice the absence of the ARB in the figure. In fact, the multi-version data cache functions very similar to the ARB, but in a distributed fashion. By keeping different versions of a memory location in the local data caches, the multi-version data cache also performs memory renaming, just like the ARB. Further work is needed to iron out the protocol for handling memory references with a multi-version data cache.

9.2.5. Multiscalar-based Multiprocessing

It is worthwhile to study how multiscalar processors can be connected together as a multiprocessor system to exploit coarse-grain parallelism. An important question that needs to be answered is

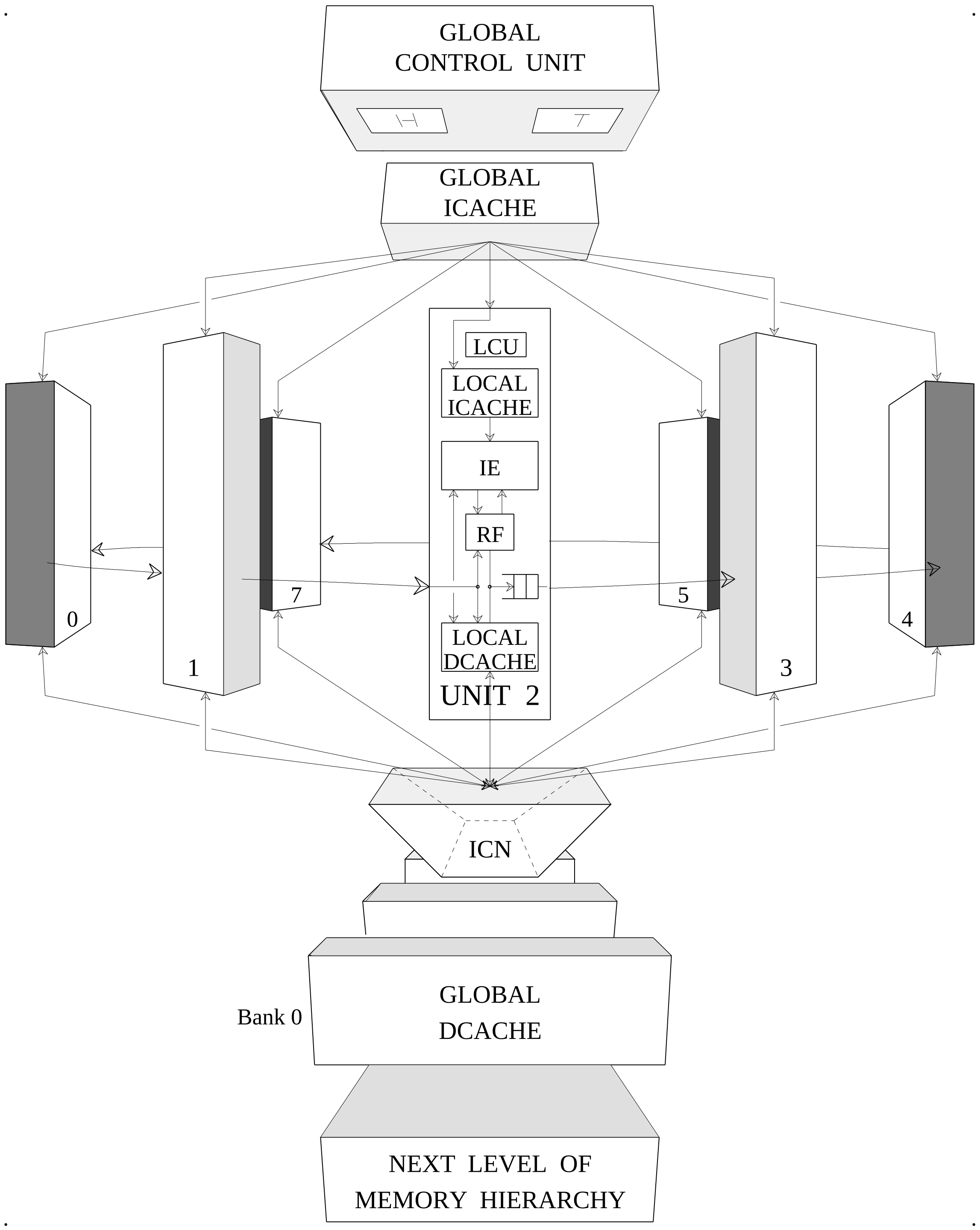


Figure 9.1. Block Diagram of an 8-Unit Multiscalar Processor with Multi-Version Data Cache

the memory model of such a multiprocessor. Will it provide sequential consistency? If not, then some sort of weak consistency? Or processor consistency? At synchronization points, each multiscalar processor can be brought to a quiescent state so as to flush out the ARB completely. We do not expect inter-processor synchronization events to occur very frequently in this multiprocessor, as fine-grain and medium-grain parallelism have already been exploited by the individual multiscalar nodes.

9.2.6. Testing

The last, but not the least, issue is testing. Testing needs to be done at various stages of product development. After installation, a processor needs to be tested periodically to make sure that no physical failures have crept in. Testing the multiscalar processor is relatively straightforward, as each execution unit can be tested independent of the remaining units. This is in contrast to testing other ILP processors such as superscalar processors and VLIW processors, in which the entire datapath is one monolithic structure. This advantage of the multiscalar processor stems from the decentralized nature of the multiscalar processor. Further work needs to be done to devise ways of performing off-line testing as well as on-line (concurrent) testing, and even ways of incorporating fault tolerance into the processor.

Appendix A. ARB

How to use the ARB in other processing paradigms?

The ARB mechanism proposed in chapter 6 is very general, and can be used in several execution models. The discussion of ARB in section 6.4 was based on the multiscalar paradigm as the underlying execution model. In this appendix, we demonstrate how the ARB can be used in other execution models, in particular superscalar processors and statically scheduled processors (such as VLIW processors) and small-scale multiprocessors. In dynamically scheduled processors, the ARB can be used for arbitrary reordering of memory references within a dynamic instruction window, as allowed by memory data dependencies. In statically scheduled processors, the ARB can be used for static reordering of ambiguous memory references, which cannot be disambiguated at compile time.

This appendix is organized as follows. Section A.1 describes how the ARB can be used in superscalar processors. It describes extensions for increasing the number of references the ARB can handle at a time. Section A.2 describes how the ARB can be used to do run-time disambiguation in statically scheduled processors, such as the VLIW processors. Section A.3 describes the use of an ARB in small-scale multiprocessors. Section A.4 gives the summary.

A.1. Application of ARB to Superscalar Processors

The working of the ARB for a superscalar processor is similar to the working of the global ARB in the multiscalar processor. In a superscalar processor, to record the original program order of the memory references, a *dynamic sequence number* is assigned to every memory reference encountered at run time. These sequence numbers are assigned by the Instruction Decode Unit or some other stage in the instruction pipeline, with the help of a counter that keeps track of the current sequence number. The counter is incremented each time a memory reference is encountered, and it wraps around when the count becomes n, where n is the maximum number of memory reference instructions allowed to be present in the active instruction window at any one time.

Once the sequence numbers are assigned, the memory references can proceed to the memory system in any order (as and when they are ready to proceed). Again, the ARB serves as the hardware structure for detecting out-of-order load-store and store-store references to the same memory location. But in this case, each ARB stage corresponds to a particular sequence number. Thus, a load bit will indicate if a load has been executed with the corresponding sequence number to the address in the address field, and a store bit will indicate if a store has been executed with that sequence number to the address in the address field. The ARB stages are logically configured as a circular queue so as to correspond to the circular queue nature of a sliding (or continuous) instruction window described in [159].

When a load or a store with sequence number i is executed, it is treated the same way as a load or store from execution unit i of the multiscalar processor is handled in section 6.4. When the processor retires the memory reference at the head pointer of the ARB, any load mark or store mark in the ARB stage at the head is erased immediately, and the head pointer of the ARB is moved forward by one stage. When recovery actions are initiated due to detection of out-of-sequence accesses to a memory location or detection of incorrect speculative execution, the ARB tail pointer is appropriately moved backwards, and the load mark and store mark columns corresponding to the sequence numbers stepped over by the tail pointer are also cleared immediately. The rest of the working is very similar to that described in section 6.4.

A.1.1. Increasing the Effective Number of ARB Stages

The number of stages in the ARB is a restricting factor on the size of the superscalar instruction window in which memory references can be reordered. The average size of the instruction window is upper bound by n/f_m, where n is the number of ARB stages and f_m is the fraction of instructions that are memory references. For instance, if n = 8 and f_m = 40%, the average window size is limited to 20 instructions. The number of ARB stages cannot be increased arbitrarily, because the complexity of the logic for checking for out-of-sequence accesses (c.f. section A.2 of Appendix) increases with the number of ARB stages. However, there are three ways of relaxing this restriction. All three rely on mapping multiple memory references to the same ARB stage, but they differ in the way this mapping is performed. All of them divide the sequential stream of memory references into *sequentially ordered groups*, and assign the same sequence number to all references in a group. Just like the memory disambiguation in the multiscalar processor, the memory disambiguation task then gets divided into two subtasks: (i) *intra-group memory disambiguation* and (ii) *inter-group memory disambiguation*. The three solutions use a single ARB to perform the inter-group disambiguation (each stage of this ARB corresponds to a group of memory references), but they differ in the way the intra-group memory disambiguation is performed.

Solution 1: The first solution is to consider a sequence of loads and a single store as a single group, all having the same sequence number. Distinct sequence numbers are assigned only to the stores; the loads are assigned the same sequence number as that of either the previous store or the succeeding store. This permits the stores to be ordered and the loads to be partially ordered, which is sufficient to guarantee execution correctness in a uniprocessor (it may not guarantee sequential consistency in a multiprocessor, if a processor performs out-of-order execution from a group of references). Thus intra-group memory disambiguation is avoided. This solution allows the instruction window to have as many stores as the number of ARB stages, and the upper bound on the average window size increases to n/f_s, where f_s is the fraction of instructions that are stores. Furthermore, it increases the utilization of the ARB hardware because each ARB stage can now correspond to multiple loads and one store.

Solution 2: The second solution is to divide the sequential reference stream into groups, with no attention being paid to the load/store nature of the references. In order to ensure correctness of memory accesses, the references in each group are executed in program order, thereby avoiding intra-group memory disambiguation as in solution 1. Notice that two references from different groups can be issued out-of-order, and the ARB will detect any out-of-sequence memory accesses to the same address. Solution 2 allows multiple stores to be present in the same group, and is therefore less restrictive than solution 1.

Solution 3: The third solution is to divide the sequential reference stream into groups as in solution 2, and then use the two-level hierarchical ARB described in section 6.4.5.2. This will allow the references within a group to be executed out-of-order. Hierarchical ARBs help to inflate the number of ARB stages available for keeping track of memory references. In particular, a two-level hierarchical ARB allows as many memory references in an instruction window as the sum of the number of stages in the local ARBs.

A.2. Application of ARB to Statically Scheduled Machines

In statically scheduled machines, the ARB can be used for performing additional compile-time reordering of memory references, over what is allowed by static disambiguation techniques. We do not claim ARB to be the best scheme for statically scheduled machines. Our only claim is that the ARB can enhance the performance of statically scheduled machines by allowing statically unresolved loads as well as statically unresolved stores. Remember that all dynamic disambiguation schemes proposed for statically scheduled machines have limitations, as disussed in Section 6.3.2.

A.2.1. Basic Idea

Static memory disambiguation is performed as much as possible. Whenever a set of m ($m \leq n$, where n is the number of ARB stages) memory references that can potentially conflict are to be statically reordered, these m references are grouped together as a *conflict set* (certain restrictions apply, as described in section A.2.3). To each memory reference in the conflict set, a *static sequence number*, starting from 0 to $m-1$, is assigned based on its original sequence in the program. This sequence number is kept in a small field in the instruction. Memory references that do not belong to any conflict set have a funny number, say n, as their sequence number. After assigning the sequence numbers, the memory references are statically reordered as desired.

For application in statically scheduled machines, the ARB stages are configured as a linear queue[9]. At run time, the loads and stores with sequence number n are executed as usual without any fanfare; the ARB does not come into the picture at all for them. When a load with sequence number i ($i < n$) is executed, an ARB row entry is obtained as before, and a load mark is entered in stage i of the ARB row entry. If an ARB entry had already been allotted for that address, a check is performed in the preceding ARB stages to see if a store has been executed to the same address. If so, the data stored in the value field of the nearest preceding store in the ARB row entry is forwarded to the load's destination register.

When a store with sequence number i ($i < n$) is executed, the same sequence of steps are followed, except that a check is performed in the succeeding stages to see if a load has been executed to the same address. If so, an out-of-sequence memory access is detected, and recovery actions are initiated by means of repair code.

Example A.1: We shall illustrate the working of the ARB for a statically scheduled processor through an example. Consider the same sequence of loads and stores as in Example 6.1. Assume that the static disambiguation techniques could not disambiguate these references, and that all the 4

---

[9] It is not mandatory to configure the stages as a linear queue; they can be configured as a circular queue as well. Because the size of a conflict set may often be less than the number of ARB stages, a circular queue configuration may not allow a perfect, identical mapping between the static sequence numbers assigned to the references in a conflict set and the ARB stage numbers the references map to at run time (***i.e.***, it may not map a reference with static sequence number i to ARB stage i during execution).

references are included in a conflict set. Table A.1 shows the static sequence numbers of the references. The ‘‘After Reordering’’ columns show the statically reordered code and their sequence numbers. The progression of the ARB contents during the execution of these loads and stores is same as that given in Figure 6.2; the only difference is the manner in which recovery is performed when an out-of-order load-store or store-store to a memory address is detected.

Table A.1: Example Sequence of Loads and Stores

Before Reordering		After Reordering	
Code	Sequence No.	Code	Sequence No.
STORE R2, 0(R1)	1	LOAD R3,–20(R2)	3
STORE R4, 0(R2)	2	STORE R4, 0(R2)	2
LOAD R3,–20(R2)	3	LOAD R5, 0(R3)	4
LOAD R5, 0(R3)	4	STORE R2, 0(R1)	1

A.2.2. Reclaiming the ARB Entries

Because the ARB stages are configured as a linear queue, the reclaiming of ARB entries is slightly different from that described in section 6.4.4. Previously, when the memory reference at the ARB head stage is committed, the two ARB columns that correspond to the load marks and store marks of the reference’s sequence number were cleared in a cycle. However, in this case, all the references in a conflict set should be committed at the same time. For this purpose, the ARB needs to determine the condition when the execution of all references in a conflict set are over. An easy way of determining this is to check if every memory reference with sequence number ≤ the maximum sequence number encountered by the ARB so far from the current conflict set have been executed[10].

When it is determined that all the references from a conflict set have been executed, all the store values stored in the value fields are forwarded to the data cache / main memory. If there are multiple store values to the same address, the one with the highest sequence number alone is forwarded. The entire ARB is then cleared in one cycle, and now the ARB is ready to handle the next conflict set. It is possible that there may be many store values to be forwarded to the data cache / main memory, causing a traffic burst, which might stall the processor for a few cycles. One simple solution to avoid this stall is to partition the ARB stages into two as shown in Figure A.1, and use the second partition when the first partition is busy forwarding values, and vice versa. When a load is executed, it can

		Partition 1				Partition 2 (Current Partition)			
	Address	Stage 0	Stage 1	Stage 2	Stage 3	Stage 0	Stage 1	Stage 2	Stage 3
Bank 0	2000	0 0	0 0	0 **1** 50	0 0	0 0	**1** 0	0 0	0 0
		0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0
		0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0
		0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0
Bank 1	2048	0 0	0 **1** 10	0 0	0 0	0 0	0 0	0 0	0 0
		0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0
		0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0
		0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0

Figure A.1: A Two-Way Partitioned, 4-Stage ARB

[10] This assumes that the last member of the conflict set, the one with sequence number $m-1$, got reordered, and is no longer the last memory reference in its conflict set in the reordered instruction stream. If this is indeed not the case, the memory reference numbered $m-1$ can be excluded from the conflict set at compile time.

search for earlier stores in both partitions of its row entry so that it gets the correct memory value from the ARB (if the value is present), and not the stale value from the data cache / main memory. In Figure A.1, Partition 2 is the current partition. So when the load with sequence number 1 (corresponding to ARB stage 1) to address 2000 is executed, the request searches in stage 0 of Partition 2 and all stages of Partition 1 for a previous store to the same address. It obtains the correct value 50 from Stage 2 of Partition 1, and therefore the load request is not forwarded to the data cache / main memory.

A.2.3. Restrictions to Reordering of Ambiguous References

The ARB allows the static reordering of all *unambiguous references* in a program. There are some restrictions to the reordering of *ambiguous references*, however. These restrictions are all imposed by the finite size of a conflict set. If the ARB is used as discussed above, the size of a conflict set is limited by the number of ARB stages. In order to relax this restriction, any of the three techniques discussed in section A.1.1 can be used. For statically scheduled processors, we have one more solution available, which is to allow multiple conflict sets to overlap in the reordered static instruction stream. Multiple conflict sets can be allowed to overlap as long as it is known that no two references from distinct overlapping conflict sets will ever conflict. (This implies that a memory reference can belong to at most one conflict set.) Overlapping of conflict sets allows further code reordering and increases the utilization of the ARB hardware because each ARB stage can now correspond to multiple memory references. For purposes of termination-determination of the overlapping conflicting sets, the references in each conflict set can be tagged with a distinct tag so that the termination-determination mechanism can separately determine the termination of each conflict set in the overlapping conflict sets. It is also possible to use separate, independent ARBs to handle each conflict set in the overlapping conflict sets.

Whatever solutions and combinations thereof we use for relaxing the restriction imposed by the finite number of ARB stages, we have to pay the price while dealing with re-entrant code such as loops and recursive procedure calls. This is because each memory reference in the static re-entrant code may map to multiple memory references at run time, and in general it is not possible to statically determine the number of memory references it maps to at run time. Thus, a conflict set cannot in general extend beyond an iteration of a loop or an instantiation of a procedure call. The restrictions can be formally stated as follows: in general it is not possible to reorder a pair of ambiguous references m_1 and m_2 of the following categories: (i) m_1 is inside a re-entrant code and m_2 is outside the re-entrant code, and (ii) m_1 and m_2 are separated by a re-entrant code (*i.e.*, m_1 precedes the re-entrant code and m_2 succeeds the re-entrant code) that has at least one memory reference m_3 belonging to some conflict set. In the first case, m_1 could map to multiple references m_1^i at run-time, and all of $\{m_1^i, m_2\}$ have to be in the same conflict set or overlapping conflict sets, whose size could be much too large for the ARB to support. In the second case, m_3 could map to multiple references m_3^i at run-time, and all of $\{m_1, m_3^i, m_2\}$ have to be in the same conflict set or overlapping conflict sets, which again could be much too large for the ARB to support. It is important to note that the ARB allows the reordering of two ambiguous references m_1 and m_2 within a re-entrant code, even if they are separated by other ambiguous references.

Example A.2: The restrictions on what can be reordered and what cannot be reordered can be best illustrated by an example. Consider the piece of machine-level code in Figure A.2, in which M1 through M5 denote memory references.

```
        M1
        ...
  Loop: ...
        M2
        ...
        M3
        ...
        M4
        ...
        Cond Branch
        ...
        M5
```

Figure A.2: Example to Demonstrate Restrictions to Code Motion

Assume that all the references can potentially conflict with each other. In this example, {M2, M3, M4} can form one conflict set; M1 and M5 cannot be part of this conflict set because M2, M3, and M3 belong to a loop. The only references that can be reordered with respect to each other in this example are M2, M3, and M4.

A.3. Application of ARB to Shared-memory Multiprocessors

ARBs can used to achieve two purposes in shared-memory multiprocessors: (i) detection of data races and (ii) implicit synchronization. We shall discuss them separately.

A.3.1. Detection of Data Races

When the individual processors in a multiprocessor system perform dynamic reordering of memory references, sequential consistency may be violated in the multiprocessor unless special care is taken [42]. The ARB can be used to detect potential violations to sequential consistency (by detecting data races as and when they occur) in multiprocessors consisting of dynamically scheduled processor nodes. We shall briefly describe how this can be done.

The key idea is for each processor in the multiprocessor to have a private ARB, which works similar in principle to the one described in Section A.1. Each of these ARBs hold information pertaining to the memory references from the active execution window of its processor. When a store is committed from the ARB of a processor, in addition to sending the store value to the memory system as before, the ARBs of all other processors are also checked to see if the same address is present in them. If the address is present in another ARB, then it signifies the presence of a data race. If each processor node has a private data cache, then the store values are sent to the processor's private data cache, and only the data cache traffic created by the cache consistency protocol need be sent to the other ARBs. The use of ARBs for the detection of data races is similar to the use of *detection buffers* proposed by Gharachorloo and Gibbons [61].

Example A.3: Consider the two pseudo-assembly program segments given in Table A.2 to be executed concurrently by two processors P1 and P2 in a shared-memory multiprocessor. Initially, location 100 contains 30 and location 120 contains 20. If the execution in the multiprocessor obeys sequential consistency, then it is not possible for *both* the loads in Table A.2 to fetch the initial values in locations 100 and 120, namely 30 and 20.

Figure A.3 shows how out-of-order execution in the processors (both the processors execute the load before the store) results in a violation of the sequential consistency, and how the ARBs detect this violation. The first two figures in Figure A.3 depict the execution of the two loads from the two processors. When the load from P1 is executed, a new ARB row entry is allotted for address 100 in P1’s ARB. The load request is forwarded to the memory, and a value of 30 is obtained. Similarly, when the load from P2 is executed, a new ARB row entry is allotted for address 120 in P2’s ARB. This request is also forwarded to the memory, and a value of 20 is obtained. Next the stores are executed as depicted in the subsequent two figures of Figure A.3. When the store from P1 is committed, the store value is sent to memory. The store address 120 is then forwarded to the remaining ARBs including P2’s ARB by the memory. When the entries in P2’s ARB are checked for address 120, a data race is detected. Similarly, when the store from P2 is is committed, the store address 100 is forwarded to the remaining ARBs including P1’s ARB. When the entries in P1’s ARB are checked for address 100, another data race is detected.

A.3.2. Implicit Synchronization

Another use of ARB in multiprocessors is as a rendezvous for implicit synchronization. When parallelizing sequential code into multiple tasks statically, synchronization operations can be avoided if it is known that the tasks are independent. If tasks are known to be dependent (e.g. FORALL loops and DOACR loops [122]), then synchronization operations can be added to parallelize the code. However, there are several loops and other code in which it cannot be statically determined if the tasks are dependent or independent, because of ambiguous data dependencies through memory. As

Table A.2: Example Sequence of Loads and Stores Involving Data Races

Processor P1			
Code	Seq. No.	Exec. Order	Store Value
STORE R1, 120	0	2	50
LOAD R2, 100	1	1	

Processor P2			
Code	Seq. No.	Exec. Order	Store Value
STORE R1, 100	0	2	75
LOAD R2, 120	1	1	

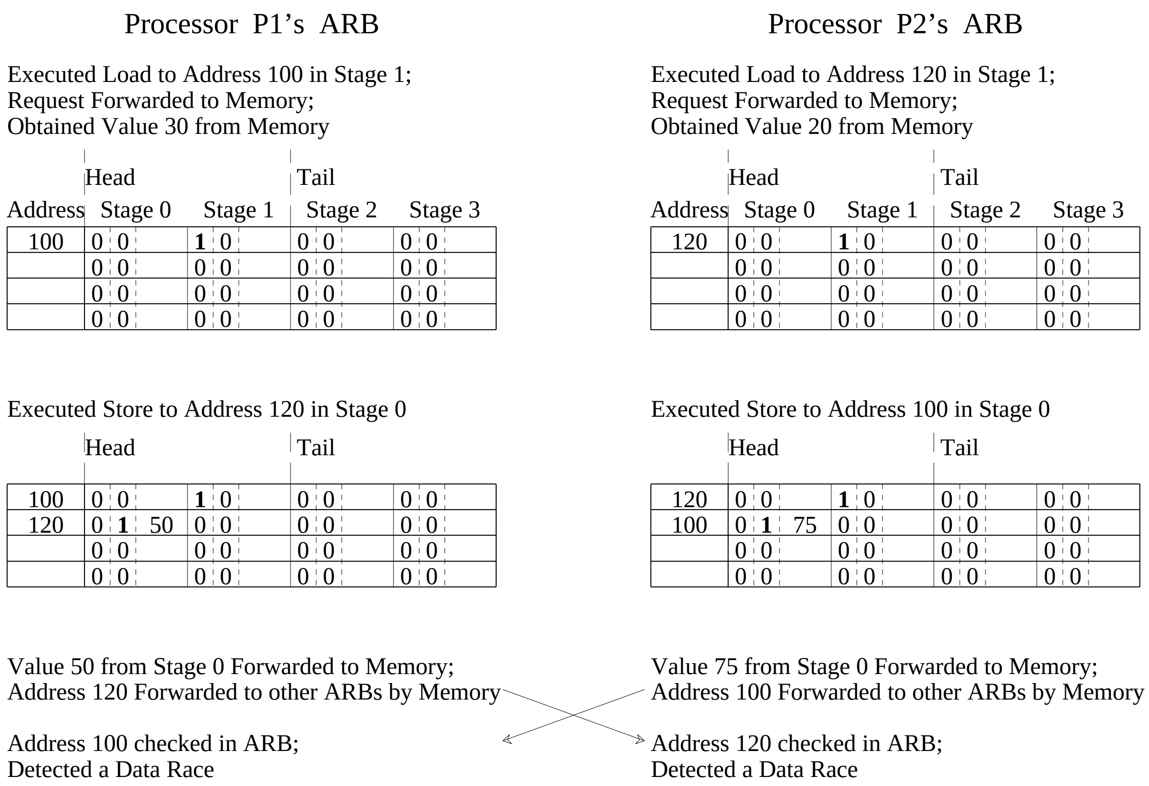


Figure A.3: ARB Contents After the Execution of Each Memory Reference in Table A.2

an example, consider the C program loop shown in Figure A.4, whose iterations have potential data dependencies because variable `j` can have the same value in multiple iterations. A compiler cannot determine if the iterations are dependent or independent, and hence it cannot parallelize the loop without using synchronization operations. Undue synchronization operations affect the performance of a multiprocessor.

```
for  i := lb, ub  do
{
     j := A[i];
     B[j] := B[j] + 1;
     ...
}
```

Figure A.4: A Loop with Potential Inter-Iteration Dependencies

The ARB can be used as an implicit synchronization mechanism in a (small-scale) shared-memory multiprocessor to aid the parallelization of loops and other code that have potential data dependencies. In particular, multiple iterations of such a loop can be issued to multiple processors in the multiprocessor, and in the event of an actual memory hazard occurring at run time, the ARB can detect the hazard.

For implicit synchronization between multiple tasks, a single global ARB is used for the entire multiprocessor. (If the individual processors perform out-of-order execution of their memory references, then in addition each processor can use a local ARB if desired, to perform intra-processor memory disambiguation.) The processors are logically ordered in some sequential order; tasks are allocated to the sequentially ordered processors as per the sequential order in which they appear in the original program. (Notice that this restriction comes about because when performing implicit synchronization, a task can be committed only when it is known for sure that all sequentially preceding tasks have completed their execution.) The global ARB has as many stages as the number of processors, and the stages have a one-to-one correspondence to the sequentially ordered processors. The load bits and store bits of a stage indicate the loads and stores that have been executed from the corresponding processor. Now, the multiprocessor system works as a multiscalar processor, with each processor of the multiprocessor corresponding to each execution unit of the multiscalar processor. It is assumed that there is facility in each multiprocessor processor node to restart the task allocated to it, if required.

We place a restriction on the number of processors in the multiprocessor, mainly because the complexity of the logic for checking for out-of-sequence accesses (c.f. section A.2 of Appendix) increases with the number of ARB stages. Notice that the traffic to be handled by the global ARB could also be a restriction; however, this restriction can be relaxed if ambiguous references are tagged, and only the ambiguous references are allowed to proceed to the global ARB. Another solution is to send to the global ARB only those requests that spill out of the cache coherency protocol. These two solutions also reduce the chances of the ARB banks getting filled with memory references.

A.4. Summary

The ARB mechanism is very general, and can be used in a wide variety of execution models. This chapter has shown how the ARB can be used in different processing models. First, we showed how the ARB can be used in a superscalar processor. The ARB can also be used in statically scheduled processors such as the VLIW for reordering ambiguous memory references that cannot be disambiguated at compile time. In this light, the ARB can be used for ordering memory references in small-scale multiprocessors too. In particular, it allows multiple iterations of a loop (having potential memory dependencies) to be issued to multiple processors, and in the event of an actual hazard occurring at run time, the ARB can detect the hazard.

REFERENCES

[1] *CDC Cyber 200 Model 205 Computer System Hardware Reference Manual*. Arden Hills, MN: Control Data Corporation, 1981.

[2] *Cray Computer Systems: Cray X-MP Model 48 Mainframe Reference Manual*. Mendota Heights, MN: Cray Research, Inc., HR-0097, 1984.

[3] *Cray Computer Systems: Cray-2 Hardware Reference Manual*. Mendota Heights, MN: Cray Research, Inc., HR-2000, 1985.

[4] R. D. Acosta, J. Kjelstrup, and H. C. Torng, ‘‘An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors,’’ *IEEE Transactions on Computers*, vol. C-35, pp. 815-828, September 1986.

[5] S. V. Adve and M. Hill, ‘‘Weak Ordering – A New Definition,’’ *Proceedings of 17th Annual International Symposium on Computer Architecture*, pp. 2-14, 1990.

[6] S. V. Adve and M. Hill, ‘‘A Unified Formalization of Four Shared-Memory Models,’’ Technical Report #1051a, Computer Sciences Department, University of Wisconsin-Madison, September 1992.

[7] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[8] A. Aiken and A. Nicolau, ‘‘Perfect Pipelining: A New Loop Parallelization Technique,’’ in *ESOP ’88: 2nd European Symosium on Programming (Nancy, France)*, Lecture Notes in Computer Science, No. 300, pp. 221-235. Springer-Verlag, New York, June 1988.

[9] A. Aiken and A. Nicolau, ‘‘A Development Environment for Horizontal Microcode,’’ *IEEE Transactions on Software Engineering*, pp. 584-594, May 1988.

[10] A. Aiken and A. Nicolau, ‘‘Optimal Loop Parallelization Technique,’’ *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 308-317, 1988.

[11] J. R. Allen, ‘‘Dependence Analysis for Subscripted Variables and Its Application to Program Transformation,’’ PhD thesis, Department of Mathematical Science, Rice University, 1983.

[12] J. R. Allen and K. Kennedy, ‘‘Automatic Loop Interchange,’’ *Proceedings of the ACM SIGPLAN 84 Symposium on Compiler Construction*, pp. 233-246, 1984.

[13] R. Allen, K. Kennedy, C. Porterfield, and J. Warren, ‘‘Conversion of Control Dependence to Data Dependence,’’ *Proceedings of 10th Annual ACM Symposium on Principles of Programming Languages*, 1983.

[14] R. Allen and K. Kennedy, ‘‘Automatic Translation for FORTRAN Programs to Vector Form,’’ *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 491-542, October 1987.

[15] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, ‘‘The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling,’’ *IBM Journal of Research and Development*, pp. 8-24, January 1967.

[16] M. Annartone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu, and J. A. Webb, ‘‘The Warp Computer: Architecture, Implementation and Performance,’’ *IEEE Transactions on Computers*, vol. C-36, pp. 1523-1538, December 1987.

[17] Arvind, K. P. Gostelow, and W. E. Plouffe, ‘‘An Asynchronous Programming Language and Computing Machine,’’ Technical Report TR 114a, Department of Information and Computation Science, University of California, Irvine, December 1978.

[18] T. M. Austin and G. S. Sohi, ‘‘Dynamic Dependency Analysis of Ordinary Programs,’’ *Proceedings of 19th Annual International Symposium on Computer Architecture*, pp. 342-351, 1992.

[19] U. Banerjee, ‘‘Speedup of ordinary Programs,’’ Tech. Report UIUCDSR-79-989, Dept. of Computer Science, Univ. of Illinois, October 1979.

[20] U. Banerjee, *Dependence Analysis for Supercomputing*. Boston, MA: Kluwer Academic Publishers, 1988.

[21] J. S. Birnbaum and W. S. Worley, ‘‘Beyond RISC: High-Precision Architecture,’’ *Proceedings of COMPCON86*, pp. 40-47, March 1986.

[22] L. J. Boland, G. D. Granito, A. U. Marcotte, B. U. Messina, and J. W. Smith, ‘‘The IBM System/360 Model 91: Storage System,’’ *IBM Journal*, pp. 54-68, January 1967.

[23] S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb, ‘‘Supporting Systolic and Memory Communication in iWarp,’’ *The 17th Annual Symposium on Computer Architecture*, pp. 70-81, 1990.

[24] R. Buehrer and K. Ekanadham, ‘‘Incorporating Dataflow Ideas into von Neumann Processors for Parallel Execution,’’ *IEEE Transactions on Computers*, vol. C-36, pp. 1515-1522, December 1987.

[25] M. Butler, T. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, ‘‘Single Instruction Stream Parallelism Is Greater than Two,’’ *Proceedings of 18th Annual International Symposium on Computer Architecture*, pp. 276-286, 1991.

[26] R. G. G. Cattell and Anderson, ‘‘Object Oriented Performance Measurement,’’ in *Proceedings of the 2nd International Workshop on OODBMS.* September 1988.

[27] P. P. Chang and W. W. Hwu, ‘‘Trace Selection for Compiling Large C Application Programs to Microcode,’’ in *Proceedings of 21st Annual Workshop on Microprogramming and Microarchitecture*, San Diego, CA, pp. 21-29, November 1988.

[28] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, ‘‘IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors,’’ *Proceedings of 18th International Symposium on Computer Architecture*, pp. 266-275, 1991.

[29] A. E. Charlesworth, ‘‘An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family,’’ *IEEE Computer*, vol. 14, September 1981.

[30] D. R. Chase, M. Wegman, and F. K. Zadeck, ‘‘Analysis of Pointers and Structures,’’ *Proceedings of ACM SIGPLAN ’90 Conference on Programming Language Design and Implementation*, pp. 296-310, 1990.

[31] W. Y. Chen, S. A. Mahlke, W. W. Hwu, and T. Kiyohara, ‘‘Personal Communication,’’ 1992.

[32] J. R. Coffman, *Computer and Job-Shop Scheduling Theory*. New York: John Wiley and Sons, 1976.

[33] E. U. Cohler and J. E. Storer, ‘‘Functionally Parallel Architectures for Array Processors,’’ *IEEE Computer*, vol. 14, pp. 28-36, September 1981.

[34] R. P. Colwell, R. P. Nix, J. J. O’Donnell, D. B. Papworth, and P. K. Rodman, ‘‘A VLIW Architecture for a Trace Scheduling Compiler,’’ *IEEE Transactions on Computers*, vol. 37, pp. 967-979, August 1988.

[35] R. Comerford, ‘‘How DEC Developed Alpha,’’ *IEEE Spectrum*, vol. 29, pp. 43-47, July 1992.

[36] D. E. Culler and Arvind, ‘‘Resource Requirements of Dataflow Programs,’’ *Proceedings of 15th Annual International Symposium on Computer Architecture*, pp. 141-150, 1988.

[37] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek, ‘‘Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine,’’ *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp. 164-175, 1991.

[38] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, ‘‘An Efficient Method of Computing Static Single Assignment Form,’’ *Conference Record of the 16th Annual Symposium on Principles of Programming Languages*, pp. 25-35, January 1989.

[39] J. Dennis, ‘‘Data Flow Supercomputers,’’ *IEEE Computer*, pp. 48-56, November 1980.

[40] K. Diefendorff and M. Allen, ‘‘Organization of the Motorola 88110 Superscalar RISC Microprocessor,’’ *IEEE Micro*, April 1992.

[41] M. Dubois, C. Scheurich, and F. Briggs, ‘‘Memory Access Buffering in Multiprocessors,’’ *Proceedings of 13th Annual Symposium on Computer Architecture*, pp. 434-442, June 1986.

[42] M. Dubois and C. Scheurich, ‘‘Memory Access Dependencies in Shared-Memory Multiprocessor,’’ *IEEE Transactions on Software Engineering*, vol. SE-16, pp. 660-673, June 1990.

[43] H. Dwyer, ‘‘A Multiple, Out-of-Order, Instruction Issuing System for Superscalar Processors,’’ Ph.D. Thesis, School of Electrical Engineering, Cornell University, 1991.

[44] H. Dwyer and H. C. Torng, ‘‘An Out-of-Order Superscalar Processor with Speculative Execution and Fast, Precise Interrupts,’’ *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICR-25)*, pp. 272-281, 1992.

[45] K. Ebcioglu, ‘‘A Compilation Technique for Software Pipelining of Loops with Conditional Jumps,’’ *Proceedings of the 20th Annual Workshop on Microprogramming (Micro 20)*, pp. 69-79, 1987.

[46] K. Ebcioglu, ‘‘Some Design Ideas for a VLIW Architecture for Sequential-natured Software,’’ *Parallel Processing (Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing)*, pp. 3-21, 1988.

[47] K. Ebcioglu and T. Nakatani, ‘‘A new Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture,’’ *Proceedings of Second Workshop on Languages and Compilers for Parallel Computing*, pp. 213-229, 1989.

[48] C. Eisenbeis, ‘‘Optimization of Horizontal Microcode Generation for Loop Structures,’’ *International Conference on Supercomputing,* pp. 453-465, 1988.

[49] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*. The MIT Press, 1986.

[50] P. G. Emma, J. W. Knight, J. H. Pomerene, R. N. Rechtschaffen, and F. J. Shapiro, ‘‘Posting Out-of-Sequence Fetches,’’ *United States Patent 4,991,090*, February 5, 1991.

[51] J. Ferrante, K. Ottenstein, and J. Warren, ‘‘The Program Dependence Graph and Its Use in Optimization,’’ *ACM Transactions on Programming Languages and Systems*, pp. 319-349, July 1987.

[52] C. N. Fischer and R. J. LeBlanc, Jr., *Crafting A Compiler*. Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc., 1988.

[53] J. A. Fisher, ‘‘Trace Scheduling: A Technique for Global Microcode Compaction,’’ *IEEE Transactions on Computers*, vol. C-30, pp. 478-490, July 1981.

[54] J. A. Fisher, ‘‘Very Long Instruction Word Architectures and the ELI-512,’’ *Proceedings of 10th Annual Symposium on Computer Architecture*, pp. 140-150, June 1983.

[55] J. A. Fisher, ‘‘The VLIW Machine: A Multiprocessor for Compiling Scientific Code,’’ *IEEE Computer,* pp. 45-53, July 1984.

[56] J. A. Fisher, ‘‘A New Architecture for Supercomputing,’’ *Digest of Papers, COMPCON Spring 1987,* pp. 177-180, February 1987.

[57] C. C. Foster and E. M. Riseman, ‘‘Percolation of Code to Enhance Parallel Dispatching and Execution,’’ *IEEE Transactions on Computers*, vol. C-21, pp. 1411-1415, December 1972.

[58] M. Franklin and G. S. Sohi, ‘‘The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism,’’ *Proceedings of 19th Annual International Symposium on Computer Architecture,* pp. 58-67, 1992.

[59] M. Franklin and G. S. Sohi, ‘‘Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors,’’ *Proceedings of The 25th Annual International Symposium on Microarchitecture (MICRO-25),* pp. 236-145, December 1992.

[60] P. P. Gelsinger, P. A. Gargini, G. H. Parker, and A. Y. C. Yu, ‘‘Microprocessors circa 2000,’’ *IEEE Spectrum*, vol. 26, pp. 43-47, October 1989.

[61] K. Gharachorloo and P. B. Gibbons, ‘‘Detecting Violations of Sequential Consistency,’’ *3rd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA’91),* pp. 316-326, 1991.

[62] G. Goff, K. Kennedy, and C.-W. Tseng, ‘‘Practical Dependence Testing,’’ *Proceedings of 1991 SIGPLAN Symposium on Compiler Construction*, pp. 15-29, 1991.

[63] J. R. Goodman, J. T. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schecter, and H. C. Young, ‘‘PIPE: a Decoupled Architecture for VLSI,’’ *Proceedings of 12th Annual Symposium on Computer Architecture*, pp. 20-27, June 1985.

[64] J. R. Goodman and P. J. Woest, ‘‘The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor,’’ *Proceedings of 15th Annual Symposium on Computer Architecture*, pp. 422-431, June 1988.

[65] J. R. Goodman, ‘‘Cache Consistency and Sequential Consistency,’’ Technical Report No. 61, SCI Committee, March 1989.

[66] G. F. Grohoski, ‘‘Machine Organization of the IBM RISC System/6000 processor,’’ *IBM Journal of Research and Development,* vol. 34, pp. 37-58, January 1990.

[67] L. J. Hendren and G. R. Gao, ‘‘Designing Programming Languages for Analyzability: A Fresh Look at Pointer Data Structures,’’ *Proceedings of 4th IEEE International Conference on Computer Languages*, pp. 242-251, 1992.

[68] L. J. Hendren, J. Hummel, and A. Nicolau, ''Abstractions for Recursive Pointer Data Structure: Improving the Analysis and Transformatio ns of Imperative Programs,'' *Proceedings of SIGPLAN'92 Conference on Programming Language Design and Implementation*, pp. 249-260, 1992.

[69] M. D. Hill and A. J. Smith, ''Evaluating Associativity in CPU Caches,'' *IEEE Transactions on Computers*, vol. 38, December 1989.

[70] S. Horwitz, P. Pfeiffer, and T. Reps, ''Dependence Analysis for Pointer Variables,'' *Proceedings of SIGPLAN'89 Symposium on Compiler Construction*, pp. 28-40, July 1989. Published as SIGPLAN Notices Vol. 24, Num. 7.

[71] P. Y. T. Hsu and E. S. Davidson, ''Highly Concurrent Scalar Processing,'' *Proceedings of 13th Annual Symposium on Computer Architecture*, pp. 386-395, June 1986.

[72] J. Hummel, L. J. Hendren, and A. Nicolau, ''Applying an Abstract Data Structure Description Approach to Parallelizing Scientific Pointer Programs,'' *Proceedings of 1992 International Conference on Parallel Processing*, vol. II, pp. 100-104, 1992.

[73] W. W. Hwu and Y. N Patt, ''HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality,'' *Proceedings of 13th Annual Symposium on Computer Architecture*, pp. 297-307, 1986.

[74] W. W. Hwu and Y. N. Patt, ''Checkpoint Repair for High-Performance Out-of-Order Execution Machines,'' *IEEE Transactions on Computers*, vol. C-36, pp. 1496-1514, December 1987.

[75] W. W. Hwu, T. M. Conte, and P. P. Chang, ''Comparing Software and Hardware Schemes for Reducing the Cost of Branches,'' *Proceedings of 16th International Symposium on Computer Architecture*, pp. 224-233, 1989.

[76] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, ''The Superblock: An Effective Technique for VLIW and Superscalar Compilation,'' *(To appear in) Journal of Supercomputing*, 1993.

[77] R. A. Iannucci, ''Toward a Dataflow / von Neumann Hybrid Architecture,'' *Proceedings of 15th Annual International Symposium on Computer Architecture*, pp. 131-140, 1988.

[78] W. M. Johnson, *Superscalar Microprocessor Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1990.

[79] R. Jolly, ''A 9-ns 1.4 Gigabyte/s, 17-Ported CMOS Register File,'' *IEEE Journal of Solid-State Circuits*, vol. 26, pp. 1407-1412, October 1991.

[80] N. P. Jouppi and D. W. Wall, ''Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines,'' *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pp. 272-282, 1989.

[81] D. R. Kaeli and P. G. Emma, ''Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns,'' *Proceedings of 18th Annual International Symposium on Computer Architecture*, pp. 34-42, 1991.

[82] G. Kane, *MIPS R2000 RISC Architecture*. Englewood Cliffs, New Jersey: Prentice Hall, 1987.

[83] S. W. Keckler and W. J. Dally, ''Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism,'' *Proceedings of 19th International Symposium on Computer Architecture*, pp. 202-213, 1992.

[84] R. M. Keller, ''Look-Ahead Processors,'' *ACM Computing Surveys*, vol. 7, pp. 66-72, December 1975.

[85] D. J. Kuck, A. H. Sameh, R. Cytron, A. V. Veidenbaum, C. D. Polychronopoulos, G. Lee, T. McDaniel, B. R. Leasure, C. Beckman, J. R. B. Davies, and C. P. Kruskal, ‘‘The Effects of Program Restructuring, Algorithm Change, and Architecture Choice on Program performance,’’ *Proceedings of the 1984 International Conference on Parallel Processing*, pp. 129-138, 1984.

[86] D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sahmeh, ‘‘Parallel Supercomputing Today and the Cedar Approach,’’ *Science*, vol. 231, February 1986.

[87] M. Kuga, K. Murakami, and S. Tomita, ‘‘DSNS (Dynamically-hazard-resolved, Statically-code-scheduled, Nonuniform Superscalar): Yet Another Superscalar Processor Architecture,’’ *Computer Architecture News*, vol. 19, pp. 14-29, June 1991.

[88] H. T. Kung, ‘‘Why Systolic Architectures?,’’ *IEEE Computer*, vol. 17, pp. 45-54, July 1984.

[89] M. S. Lam, ‘‘Software Pipelining: An Effective Scheduling Technique for VLIW Machines,’’ *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 318-328.

[90] M. S. Lam and R. P. Wilson, ‘‘Limits of Control Flow on Parallelism,’’ *Proceedings of 19th Annual International Symposium on Computer Architecture*, pp. 46-57, 1992.

[91] L. Lamport, ‘‘How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs,’’ *IEEE Transactions on Computers*, vol. C-28, pp. 241-248, September 1979.

[92] J. R. Larus and P. N. Hilfinger, ‘‘Detecting Conflicts Between Structure Accesses,’’ *Proceedings of SIGPLAN’88 Symposium on Compiler Construction*, pp. 21-34, July 1988. Published as SIGPLAN Notices Vol. 23, Num. 7.

[93] J. K. F. Lee and A. J. Smith, ‘‘Branch Prediction Strategies and Branch Target Buffer Design,’’ *IEEE Computer*, vol. 17, pp. 6-22, January 1984.

[94] J. M. Levesque and J. W. Williamson, *A Guidebook to Fortran on Supercomputers.* San Diego, California: Academic Press, Inc., 1989.

[95] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker, ‘‘Sentinel Scheduling for VLIW and Superscalar Processors,’’ *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS-V),* pp. 238-247, 1992.

[96] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, ‘‘Effective Compiler Support for Predicated Execution using the Hyperblock,’’ *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO-25),* pp. 45-54, 1992.

[97] D. E. Maydan, J. L. Hennessy, and M. S. Lam, ‘‘Efficient and Exact Data Dependence Analysis,’’ *Proceedings of 1991 SIGPLAN Symposium on Compiler Construction*, pp. 1-14, 1991.

[98] S. McFarling and J. Hennessy, ‘‘Reducing the Cost of Branches,’’ *Proceedings of 13th Annual Symposium on Computer Architecture*, pp. 396-304, 1986.

[99] K. Miura and K. Uchida, ‘‘FACOM Vector Processor VP-100/VP-200,’’ in *Proc. NATO Advanced Research Workshop on High-Speed Computing*, Julich, Germany, Springer-Verlag, June 20-22, 1983.

[100] S-M. Moon and K. Ebcioglu, ‘‘An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors,’’ *Proceedings of The 25th Annual International Symposium on Microarchitecture (MICRO-25),* pp. 55-71, 1992.

[101] K. Murakami, N. Irie, M. Kuga, and S. Tomita, ‘‘SIMP (Single Instruction Stream / Multiple Instruction Pipelining): A Novel High-Speed Single-Processor Architecture,’’ *Proceedings of 16th Annual Symposium on Computer Architecture*, pp. 78-85, 1989.

[102] T. Nakatani and K. Ebcioglu, ‘‘‘‘Combining’’ as a Compilation Technique for VLIW Architectures,’’ *Proceedings of the 22nd Annual International Workshop on Microprogramming and Microarchitecture (Micro 22),* pp. 43-55, 1989.

[103] T. Nakatani and K. Ebcioglu, ‘‘Using a Lookahead Window in a Compaction-Based Parallelizing Compiler,’’ *Proceedings of the 23rd Annual Workshop on Microprogramming and Microarchitecture (Micro 23),* pp. 57-68, 1990.

[104] SPEC Newsletter, vol. 1, Fall 1989.

[105] A. Nicolau, ‘‘Parallelism, Memory Anti-Aliasing and Correctness for Trace Scheduling Compilers,’’ Ph.D. Dissertation, Yale University, June 1984.

[106] A. Nicolau and J. A. Fisher, ‘‘Measuring the Parallelism Available for Very Long Instruction Word Architectures,’’ *IEEE Transactions on Computers*, vol. C-33, pp. 968-976, November 1984.

[107] A. Nicolau, ‘‘Percolation Scheduling: A Parallel Compilation Technique,’’ Technical Report TR 85-678, Department of Computer Science, Cornell University, 1985.

[108] A. Nicolau, ‘‘Run-Time Disambiguation: Coping With Statically Unpredictable Dependencies,’’ *IEEE Transactions on Computers*, vol. 38, pp. 663-678, May 1989.

[109] R. S. Nikhil and Arvind, ‘‘Can Dataflow Subsume von Neumann Computing?,’’ *Proceedings of 16th International Symposium on Computer Architecture*, pp. 262-272, 1989.

[110] R. S. Nikhil, G. M. Papadopoulos, and Arvind, ‘‘*T: A Multithreaded Massively Parallel Architecture,’’ *Proceedings of 19th Annual International Symposium on Computer Architecture*, pp. 156-167, 1992.

[111] R. R. Oehler and R. D. Groves, ‘‘IBM RISC System/6000 Processor Architecture,’’ *IBM Journal of Research and Development*, vol. 34, pp. 23-36, January 1990.

[112] S.-T. Pan, K. So, and J. T. Rahmeh, ‘‘Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation,’’ *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS-V),* pp. 76-84, 1992.

[113] G. M. Papadopoulos and D. E. Culler, ''Monsoon: An Explicit Token-Store Architecture,'' *Proceedings of 17th Annual International Symposium on Computer Architecture*, pp. 82-91, 1990.

[114] G. M. Papadopoulos and K. R. Traub, ''Multithreading: A Revisionist View of Dataflow Architectures,'' *Proceedings of 18th Annual International Symposium on Computer Architecture*, pp. 342-351, 1991.

[115] Y. N. Patt, S. W. Melvin, W. W. Hwu, and M. Shebanow, ''Critical Issues Regarding HPS, A High Performance Microarchitecture,'' in *Proceedings of 18th Annual Workshop on Microprogramming*, Pacific Grove, CA, pp. 109-116, December 1985.

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

[117] C. Peterson, J. Sutton, and P. Wiley, ''iWarp: A 100-MOPS, LIW Microprocessor for Multicomputers,'' *IEEE Micro*, pp. 26-29, 81-87, June 1991.

[118] K. Pettis and R. C. Hansen, ''Profile Guided Code Positioning,'' *Proceedings of ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 16-27, 1990.

[119] A. R. Pleszkun, G. S. Sohi, B. Z. Kahhaleh, and E. S. Davidson, ''Features of the Structured Memory Access (SMA) Architecture,'' *Digest of Papers, COMPCON Spring 1986*, March 1986.

[120] A. R. Pleszkun and G. S. Sohi, ''The Performance Potential of Multiple Functional Unit Processors,'' *Proceedings of 15th Annual Symposium on Computer Architecture*, pp. 37-44, 1988.

[121] D. N. Pnevmatikatos, M. Franklin, and G. S. Sohi, *Proceedings of the 26th Annual International Symposium on Microarchitecture (MICRO-26)*, 1993.

[122] C. D. Polychronopoulos and D. J. Kuck, ‘‘Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers,’’ *IEEE Transactions on Computers*, vol. C-36, pp. 1425-1439, December 1987.

[123] V. Popescu, M. Schultz, J. Spracklen, G. Gibson, B. Lightner, and D. Isaman, ‘‘The Metaflow Architecture,’’ *IEEE Micro*, pp. 10-13, 63-72, June 1991.

[124] B. R. Rau and C. D. Glaeser, ‘‘Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing,’’ *Proceedings of 20th Annual Workshop on Microprogramming and Microarchitecture*, pp. 183-198, 1981.

[125] B. R. Rau, C. D. Glaeser, and R. L. Picard, ‘‘Efficient Code Generation For Horizontal Architectures: Compiler Techniques and Architectural Support,’’ *Proceedings of 9th Annual Symposium on Computer Architecture*, pp. 131-139, 1982.

[126] B. R. Rau, ‘‘Cydra 5 Directed Dataflow Architecture,’’ *Digest of Papers, COMPCON Spring 1988*, pp. 106-113, February 1988.

[127] B. R. Rau, D. W. L. Yen, W. Yen, and R. Towle, ‘‘The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-offs,’’ *IEEE Computer*, vol. 22, pp. 12-35, January 1989.

[128] B. R. Rau, M. S. Schlansker, and D. W. L. Yen, ‘‘The Cydra^{TM} Stride-Insensitive Memory System,’’ in *1989 International Conference on Parallel Processing*, St. Charles, Illinois, pp. I-242-I-246, August 1989.

[129] B. R. Rau and J. A. Fisher, ‘‘Instruction-Level Parallel Processing: History, Overview and Perspective,’’ *The Journal of Supercomputing*, vol. 7, January 1993.

[130] Cray Research, Inc., ‘‘The Cray Y-MP Series of Computer Systems,’’ *Cray Research Inc., Publication No. CCMP-0301*, February 1988.

[131] R. M. Russel, ‘‘The CRAY-1 Computer System,’’ *CACM*, vol. 21, pp. 63-72, January 1978.

[132] G. M. Silberman and K. Ebcioglu, ‘‘An Architectural Framework for Migration from CISC to Higher Performance Platforms,’’ *Proceedings of International Conference on Supercomputing*, 1992.

[133] J. E. Smith, ‘‘A Study of Branch Prediction Strategies,’’ *Proceedings of 8th International Symposium on Computer Architecture*, pp. 135-148, May 1981.

[134] J. E. Smith, ‘‘Decoupled Access/Execute Architectures,’’ *Proceedings of 9th Annual Symposium on Computer Architecture*, pp. 112-119, April 1982.

[135] J. E. Smith, A. R. Pleszkun, R. H. Katz , and J. R. Goodman, ‘‘PIPE: A High Performance VLSI Architecture,’’ in *Workshop on Computer Systems Organization*, New Orleans, LA, 1983.

[136] J. E. Smith, S. Weiss, and N. Pang, ‘‘A Simulation Study of Decoupled Architecture Computers,’’ *IEEE Transactions on Computers*, vol. C-35, pp. 692-702, August 1986.

[137] J. E. Smith, et al, ‘‘The ZS-1 Central Processor,’’ *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, pp. 199-204, 1987.

[138] J. E. Smith and A. R. Pleszkun, ‘‘Implementing Precise Interrupts in Pipelined Processors,’’ *IEEE Transactions on Computers*, vol. 37, pp. 562-573, May 1988.

[139] J. E. Smith, ‘‘Dynamic Instruction Scheduling and the Astronautics ZS-1,’’ *IEEE Computer*, pp. 21-35, July 1989.

[140] M. D. Smith, M. Johnson, and M. A. Horowitz, ‘‘Limits on Multiple Instruction Issue,’’ *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pp. 290-302, 1989.

[141] M. D. Smith, M. S. Lam, and M. A. Horowitz, ‘‘Boosting Beyond Static Scheduling in a Superscalar Processor,’’ *Proceedings of 17th Annual Symposium on Computer Architecture*, pp. 344-354, 1990.

[142] M. D. Smith, M. Horowitz, and M. S. Lam, ‘‘Efficient Superscalar Performance Through Boosting,’’ *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp. 248-259, 1992.

[143] M. D. Smith, ‘‘Support for Speculative Execution in High-Performance Processors,’’ Ph.D. Thesis, Department of Electrical Engineering, Stanford University, November 1992.

[144] G. S. Sohi and S. Vajapeyam, ‘‘Instruction Issue Logic for High-Performance, Interruptible Pipelined Processors,’’ in *Proceedings of 14th Annual Symposium on Computer Architecture*, Pittsburgh, PA, pp. 27-34, June 1987.

[145] G. S. Sohi, ‘‘High-Bandwidth Interleaved Memories for Vector Processors - A Simulation Study,’’ Computer Sciences Technical Report #790, University of Wisconsin-Madison, Madison, WI 53706, September 1988.

[146] G. S. Sohi and S. Vajapeyam, ‘‘Tradeoffs in Instruction Format Design For Horizontal Architectures,’’ *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pp. 15-25, 1989.

[147] G. S. Sohi, ‘‘Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers,’’ *IEEE Transactions on Computers*, vol. 39, pp. 349-359, March 1990.

[148] G. S. Sohi and M. Franklin, ‘‘High-Bandwidth Data Memory Systems for Superscalar Processors,’’ *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp. 53-62, 1991.

[149] B. Su and J. Wang, ‘‘GURPR*: A New Global Software Pipelining Algorithm,’’ *Proceedings of the 24th Annual Workshop on Microprogramming and Microarchitecture*, pp. 212-216, 1991.

[150] D. J. Theis, ‘‘Special Tutorial: Vector Supercomputers,’’ *IEEE Computer*, pp. 52-61, April 1974.

[151] J. E. Thornton, ‘‘Parallel Operation in the Control Data 6600,’’ *Proceedings of AFIPS Fall Joint Computers Conference*, vol. 26, pp. 33-40, 1964.

[152] J. E. Thornton, *Design of a Computer -- The Control Data 6600*. Scott, Foresman and Co., 1970.

[153] R. M. Tomasulo, ‘‘An Efficient Algorithm for Exploiting Multiple Arithmetic Units,’’ *IBM Journal of Research and Development*, pp. 25-33, January 1967.

[154] H. C. Torng and M. Day, ‘‘Interrupt Handling for Out-of-Order Execution Processors,’’ Technical Report EE-CEG-90-5, School of Electrical Engineering, Cornell University, June 1990.

[155] A. K. Uht, ‘‘Hardware Extraction of Low-level Concurrency from Sequential Instruction Streams,’’ PhD thesis, Carnegie-Mellon University, Department of Electrical and Computer Engineering, 1985.

[156] A. K. Uht and R. G. Wedig, ‘‘Hardware Extraction of Low-level Concurrency from a Serial Instruction Stream,’’ *Proceedings of International Conference on Parallel Processing*, pp. 729-736, 1986.

[157] S. Vajapeyam, G. S. Sohi, and W.-C. Hsu, ‘‘Exploitation of Instruction-Level Parallelism in a Cray X-MP Processor,’’ in *Proceedings of International Conference on Computer Design (ICCD 90)*, Boston, MA, pp. 20-23, September 1990.

[158] A. H. Veen, ‘‘Dataflow Machine Architectures,’’ *ACM Computing Surveys*, vol. 18, pp. 365-396, December 1986.

[159] D. W. Wall, ‘‘Limits of Instruction-Level Parallelism,’’ *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp. 176-188, 1991.

[160] D. W. Wall, ‘‘Predicting Program Behavior Using Real or Estimated Profiles,’’ *Proceedings of ACM SIGPLAN ’91 Conference on Programming Language Design and Implementation*, pp. 59-70, 1991.

[161] S. Weiss and J. E. Smith, ‘‘A Study of Scalar Compilation Techniques for Pipelined Supercomputers,’’ *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS-II),* pp. 105-109, 1987.

[162] A. Wolfe and J. P. Shen, ‘‘A Variable Instruction Stream Extension to the VLIW Architecture,’’ *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV),* pp. 2-14, 1991.

[163] M. J. Wolfe, ‘‘Optimizing Supercompilers for Supercomputers,’’ Ph.D. Dissertation, Univ. of Illinois, Urbana-Champaign, October 1982.

[164] W. A. Wulf, ‘‘The WM Computer Architecture,’’ *Computer Architecture News,* vol. 16, March 1988.

[165] W. A. Wulf, ‘‘Evaluation of the WM Computer Architecture,’’ *Proceedings of 19th Annual International Symposium on Computer Architecture*, pp. 382-390, 1992.

[166] T. Y. Yeh and Y. N. Patt, ‘‘Alternative Implementations of Two-Level Adaptive Training Branch Prediction,’’ *Proceedings of 19th Annual International Symposium on Computer Architecture*, pp. 124-134, 1992.