Software Register Synchronization for Super-Scalar Processors with Partitioned Register Files

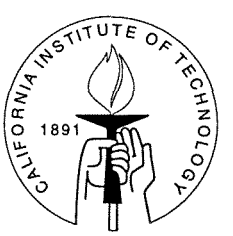
Thesis by

Daniel Maskit

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy



California Institute of Technology

Pasadena, California

1997

(Submitted December 12, 1996)

Acknowledgements

Many thanks to my advisor, Steve Taylor. Steve has taught me a great deal about parallel programming and about the practice of academia. He has encouraged me to do the work which I am interested in, allowed me to do things my way, and given me the confidence to believe in my intuition. In addition, Steve is one of the most generous people I have ever worked with. He treats his graduate students as his colleagues, and has done everything he can to make our graduate careers not just as productive, but as enjoyable, as possible.

Special thanks are also due to Bill Dally. Bill has gone out of his way to take an interest in my career. He has advised me both academically and professionally. He has offered his time and the time of others in his lab to further our common goals. I look forward to future productive work with Bill.

I would also like to thank the other members of my committee. Al Barr and Mani Chandy have participated in one way or another with almost all phases of my academic progress at Caltech. They have consistently been pleasant to work with, and obviously been interested in my progress. They have both encouraged me to broaden my horizons, and have introduced me to new and interesting ways of thinking about the world of computer science.

This work would not have been possible without the assistance and great patience of the members of the MIT Concurrent VLSI Architecture (CVA) Group. In particular Steve Keckler has taught me a great deal about architecture, VLSI, and technical writing (although this thesis reflects how much I had to learn far more than it reflects his positive influence). Others deserving of mention include Yevgeny Gurevich, Andrew Shultz, Fletcher Sandbeck, Andrew Chang, Nick Carter, and Michael Noakes.

A great debt is owed to Rich Lethin who set aside his own research to spend nearly

a year training me to the point where I could assume primary responsibility for the Multiflow compiler work. Rich's teaching skills and generosity were invaluable to my work.

In addition, all the members of the CVA group have conspired to be wonderful hosts to me during the many trips I made to Boston in the course of this work. They made me feel welcome, and treated me not so much as a visiting colleague, but as an honored guest.

While at Caltech I have continued the study of history inspired by my don from Sarah Lawrence College, Francis B. Randall. I have completed a history minor for my Ph.D. working primarily with Doug Flamming who has made this study both educational and fun.

My time in graduate school has been vastly improved both by the friends I left behind when I came to California: Jesse Lentchner and Rebecca Terner, Dirk Dawson and Laura Emerick, Bruce Musser, Ethan Galant and Leigh Hendrickson; as well as new friends I have made here at Caltech: Maneesh Sahani, Jennifer Linden, Len and Shelly Mueller, Eve Schooler and Bob Felderman.

I would also like to thank the other members of the Scalable Concurrent Programming Laboratory, both past and present: Mike Palmer, Marc Rieffel, Jerrell Watts, Bryan Chow, Yair Zadik, Dave Bourgeois, and Andy Fyfe. The lab has consistently been a good working environment, and a great group of people to socialize with.

The finest treasure that I have encountered during my time at Caltech, however, is my fiancee Kelly Smith. I look forward to many happy years with her.

I would be remiss if I did not mention my family. My father Bernie and his wife Wilma have been encouraging and supportive, and have maintained a warm household which allows me to continue thinking of New York as home.

My paternal grandmother Celia has always stood by and encouraged and made it very clear how proud she is of not only my accomplishments, but those of my brothers as well. It is a pleasure to continue to make her proud.

My mother Paula has been supportive of some of the very difficult decisions I have

had to make during my graduate career, and in the process has made her love for me obvious.

My brothers Sid and Jonathan have helped keep my feet on the right path, and provided many enjoyable days of exploring exotic places from the swamps of Louisiana to the bistros of Paris. I am extremely lucky to have siblings who are also wonderful friends.

I would finally like to acknowledge the critical support I received from my aunt, Mae Lord. Mae's untimely death has left me deeply saddened. I have been deprived of one of the finest confidants and advisors one could hope for.

The funding for the work described in this thesis has come from a variety of sources. I spent several years being funded by an NSF Graduate Fellowship, with the research project itself funded by the Advanced Research Projects Agency, ARPA Order number 8176, and monitored by the Office of Naval Research under contract number N00014-91-J-1986. The rest of the work was conducted with funding from the Advanced Research Projects Agency under contract number DABT63-95-C-0116. Some of the hardware resources used to run simulations reported on in this work were provided by the U.S. Air Force under Air Force Office of Scientific Research Grant F49620-95-1-0081. In addition, I was designated a finalist by the Hertz Foundation in 1994.

Abstract

Increases in high-end microprocessor performance are becoming increasingly reliant on simultaneous issuing of instructions to multiple functional units on a single chip. As the number of functional units increases, the chip area, wire lengths, and delays required for a monolithic register file become unreasonable. *Future microprocessors will have partitioned register files.* The correctness of contemporary super-scalar processors relies on synchronized accesses to registers. This issue will be critical in systems with partitioned register files. Current techniques for managing register access ordering, such as *register scoreboarding* and *register renaming*, are inadequate for architectures with partitioned register files. This thesis demonstrates the difficulties of implementing these techniques with a partitioned register file, and introduces a novel compiler algorithm which addresses this issue.

Whenever a processor using *register scoreboarding* or *register renaming* issues an instruction, either the scoreboard or the register name table must be accessed to check the instruction's sources and destination. If the register file is partitioned, checking the scoreboard or name table for a remote register is difficult. One functional unit cannot determine at runtime when it is safe to write to a register in another functional unit's register file. While these techniques can be supported through use of a global or partitioned scoreboard, such an implementation would be complex, and have latency problems similar to those of a monolithic register file.

This work discusses the organization of multiple functional units into loosely-coupled groups of functional units that can communicate via direct register writes, but with purely local hardware interlocks to force synchronization. A novel compiler algorithm, Software Register Synchronization (SRS), is introduced. A comparison between SRS and existing hardware mechanisms is conducted using the Multiflow compiler modified to generate code for the MIT M-Machine. Experiments to evaluate

the SRS algorithm are run on the M-Machine simulator being used for architectural verification. In order to support partitioned register file architectures, an alternative to traditional hardware methods for managing register synchronization needs to be developed. This thesis presents a novel compiler algorithm to address this need. The SRS algorithm is described, demonstrated to be correct, and evaluated. Details of the implementation of the SRS algorithm within the Multiflow compiler for the MIT M-Machine are provided.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

Increases in high-end microprocessor performance are becoming increasingly reliant on simultaneous issuing of instructions to multiple functional units on a single chip. As the number of functional units increases, the chip area, wire lengths, and delays required for a monolithic register file become unreasonable. *Future microprocessors will have partitioned register files.* The correctness of contemporary super-scalar processors relies on synchronized accesses to registers. This issue will be critical in systems with partitioned register files. Current techniques for managing register access ordering, such as *register scoreboarding* [59] and *register renaming* [60], are inadequate for architectures with partitioned register files. This work demonstrates the difficulties of implementing these techniques with a partitioned register file, and introduces a novel compiler algorithm which addresses this issue.

This thesis assumes a generic model of a partitioned register file system in which each functional unit is assigned to a local register file. More than one functional unit can be assigned to the same local register file. Functional units can read only from their local register file. All such reads access the local hardware scoreboard and are synchronizing operations. When an instruction issues, its destination register is marked EMPTY. Attempts to read an EMPTY register will stall until the register is marked FULL. Functional units can write to either their local register file, or to remote register files on the same chip. Writes do not access the scoreboard prior to issuing. The only access to the scoreboard performed by a write is to mark the destination register FULL after the write has completed.

State of the art high performance processors manage register synchronization through *register scoreboarding* or *register renaming.* When an instruction issues, ei-

ther the scoreboard or the register name table must be accessed to check its sources and its destination. If the register file is partitioned, checking the scoreboard or name table for a remote register is difficult, one functional unit cannot know when it is safe to write to a register in another functional unit's register file. While these techniques can be supported through use of a global scoreboard, or a partitioned scoreboard, they are complex to implement, and have latency problems similar to those of a monolithic register file.

An alternative to global scoreboarding or renaming is to change the model in which the functional units communicate with one another. Instead of communicating via general registers, remote register writes can be placed into a queue [18], to be read at the receiving functional unit's leisure. The receiving functional unit can then locally manage register synchronization within its own register file. This solution requires additional hardware, the queue, and can increase compiler complexity as queue write order could be critical. [18, 10].

This work discusses the organization of multiple functional units into loosely-coupled clusters. These are groups of functional units that can communicate via direct register writes, but with purely local hardware interlocks to force synchronization. A compiler algorithm, Software Register Synchronization (SRS) is introduced. A comparison between SRS and existing hardware mechanisms will be made. This comparison examines solutions for a contemporary super-scalar processor with a monolithic register file. The comparison is conducted using the Multiflow compiler modified to generate code for the MIT M-Machine. Experiments to evaluate the SRS algorithm are run on the M-Machine simulator being used for architectural verification.

1.1 Trends in Computer Architecture

The first RISC microprocessor [46, 48] heralded a new era in computer architecture. Since the advent of RISC, emphasis in microprocessor design has been on balancing VLSI complexity with compiler complexity. Figure 1.1 shows the design space for a variety of architectures showing the tradeoff between these complexities. In this new era, progress in computer architecture has been driven by architectural innovation enabled by increases in available chip area as process feature size has decreased. One of the major areas of emphasis in early RISC architectures was the use of pipelining to increase instruction throughput. This emphasis on RISC and pipelining as a replacement for earlier microcoded designs revived the concept of *static scheduling.* Rather than having a complex control unit within the processor to manage all of the aspects of resource management, *static scheduling* performs some of this management within the compiler.

The MIPS (Microprocessor without Interlocked Pipe Stages) [32] processor developed at Stanford in the early 1980's was one of the early RISC architecture projects. This processor is of particular interest since, as its name suggests, it does not have hardware interlocks in the pipeline. Relative to the Berkeley RISC project [46, 47], the MIPS architecture places a higher emphasis on processor performance than on compiler simplicity. In order to simplify the pipeline hardware, hazard detection and avoidance is assigned to the compiler.

The next major development in architecture was superscalar microprocessors. This development drew on earlier computer designs, such as the CDC-6600 [59] which had superscalar designs but required multiple chips, and the IBM 360/91 [1] which was the first machine to use *register renaming.* The first of these chips was the America research processor which became the IBM RS/6000 [30, 45]. This processor has three functional units: fixed-point, floating-point, and branch. The emphasis of this

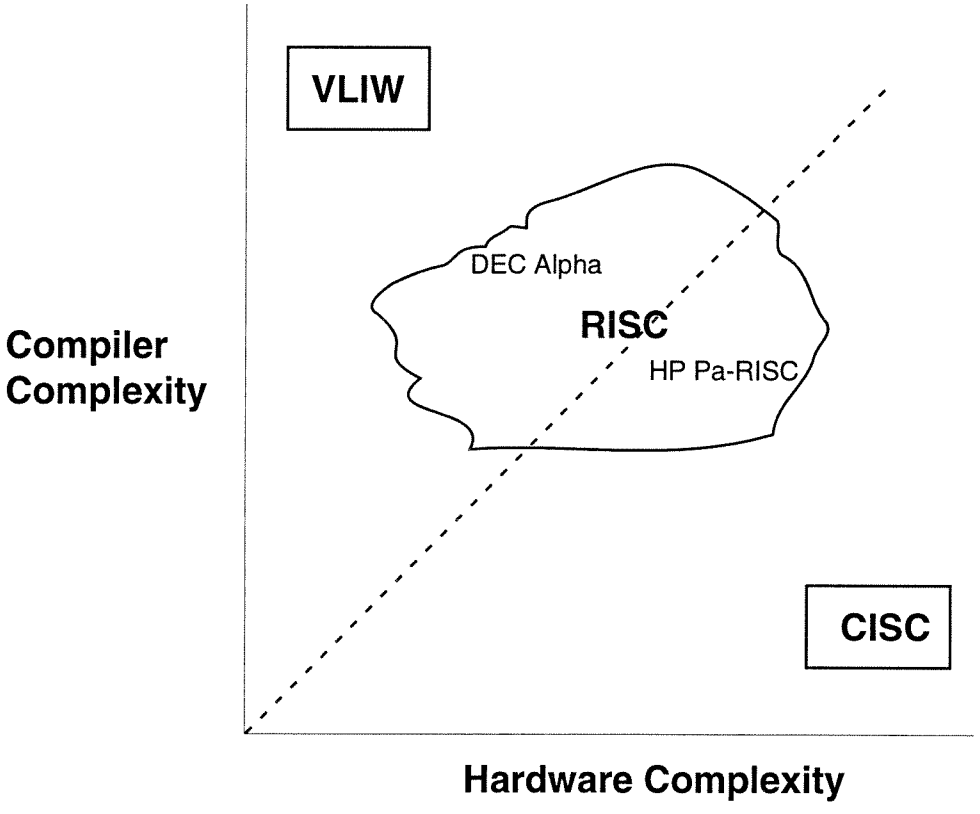


Figure 1.1: **Tradeoff Between Compiler and Hardware Complexity in Computer Designs**. This figure shows the spectrum of design space from a Complex Instruction Set Computer (CISC) like the DEC VAX which featured extensive microcoding to a VLIW machine which has all machine operation choreographed by the compiler.

design is on simplifying hardware without significantly adding to compiler complexity. One of the key ideas in this design is the separation of the register file into specific registers intended to be used by each functional unit. This functional separation reduces the circumstances in which the functional units will interfere with each other, but requires explicit compiler management of this interference when it does occur. For example, it is the compiler's responsibility to ensure that moving data from one register type to another is performed safely.

The most recent commercially available high-end processors, such as the Intel Pentium Pro [13], HP PA-8000 [27], MIPS R10000 [43], and the IBM/Motorola PowerPC 604 [44], feature up to six functional units, and hardware implementations of register synchronization techniques that allow out-of-order issuing of instructions. These machines employ complex hardware, which allows some degree of simplification within the compiler. For example, the coordination of out-of-order instruction issue is managed entirely in hardware.

An alternative approach to supporting multiple functional units is Very Long Instruction Word (VLIW) architectures. This term, coined by Josh Fisher [20], describes a machine with many functional units, and no interlocks. The compiler required for such a machine is quite complex, as it must statically schedule all of the machine's resources for each clock cycle. In such a machine, improper compiler choreography of operations can violate program correctness, or even crash the machine.

Current super-scalar designs allow reads and writes to a single register file from any one of the functional units, requiring, in general, two read ports and one write port on the register file for each functional unit. Since each write port requires one word-line for selection, and one bit-line to move the data [42], the size of the register file is a function of the square of the number of write ports. Thus, doubling the number of functional units results in a four-fold increase in the size of the register file. In addition the register file access delays increase due to parasitic capacitances:

wire parasitics due to longer bit- and word-lines as well as transistor parasitics for each port. Both area constraints and wire delays limit the practical number of register file ports.

In order to continue extracting more performance out of a single chip by adding additional functional units, it will be necessary to partition the register file. This solution will yield smaller register files and higher tolerance for delays in writing distant register files. One effect of partitioning the register file is an increase in compiler complexity as the compiler must take on responsibility for managing transfers between register files.

1.2 Register Synchronization

Register hazards result when the relative order of instructions reading or writing their registers is not strictly controlled. Read-after-write (RAW) hazards occur when an instruction issues before the instruction producing its data has completed. Write-after-read (WAR) hazards occur when an instruction writes its result to a register before an instruction that was supposed to read the old value is able to issue. Write-after-write (WAW) hazards are due to instructions completing in a different order than they were specified (or assumed) by the compiler. Thus if two sequential operations A followed by B both write into the same register $r1$, and B completes before A, then $r1$ will have the incorrect result. Any subsequent instruction that reads from that register will get an incorrect result.

In purely sequential machines, WAW and WAR (collectively called WAX) hazards do not occur, as instructions are fetched and executed in exactly the order in which they are specified in the object code. However, the performance improvements seen in modern microprocessors have opened a window for these hazards. Pipelined processors in which register writes can occur from multiple stages of the pipeline,

non-blocking memory systems, multiple arithmetic units–such as floating point or superscalar units, and out-of-order instruction issue, often performed in hardware by high performance processors, all pose a threat of WAX hazards. Some examples of modern processors that use non-blocking caches, multiple arithmetic units, deep pipelines, and out-of-order execution are the MIPS R10000 [43], HP PA-8000 [27], Intel Pentium Pro [13], and the IBM/Motorola PowerPC 604 [44].

RAW hazards can be eliminated with a relatively simple hardware scoreboard, which marks registers empty when an instruction issues, and marks it full when the result returns. Instructions which attempt to read a register marked empty are blocked from executing until the empty register is marked full. RAW hazards are "true" hazards as they indicate the data dependencies between instructions; handling them in some way in hardware is usually necessary because they are frequent. Conceptually, WAX hazards can be eliminated with an infinite number of registers. No registers would be reused, and no WAX conflicts could occur. With a limited register set, registers must be reused. Without either software or hardware precautions, register reuse can result in WAX hazards, causing the program to produce incorrect results. WAR hazards may also be eliminated by prohibiting hardware out-of-order instruction issue completely; or by preventing an operation which writes a particular register from issuing prior to the completion of any instruction that precedes it in the static schedule that reads from the same register. WAW hazards are due to out of order instruction completion, which can be caused by variable instruction latency, and require special hardware for detection. This thesis focuses on reducing hardware complexity by moving WAW detection into the compiler and converting WAW hazards into RAW hazards that can be detected by the already necessary register scoreboard.

The motivation for this algorithm comes from compiler development for the MIT M-Machine [19] currently being designed by the Concurrent VLSI Architecture Group

at MIT. The Multi-ALU processor (MAP) chip, which serves as a single processing node in the M-Machine, performs remote memory accesses asynchronously, but does not check the synchronization state of registers prior to overwriting them. This allows processing to continue while awaiting completion of a remote memory access, but does not protect against WAW hazards. This architectural decision poses a challenge to the compiler: to efficiently support a non-blocking memory system while maintaining program correctness by preventing WAW conflicts. The MAP processor of the M-Machine features loosely-coupled CPUs on a single chip. Each of these processors can write the register files of all other processors on the same chip. Traditional methods of preventing WAW hazards are particularly problematic as they require each CPU to keep track of both local and remote registers. The simplified scoreboarding hardware of the MAP, the non-deterministic memory latencies, and the multiple clusters with distinct register files all require the efficient software solution described in this thesis.

1.3 Hardware Methods for Managing Register Synchronization

The two main hardware techniques for preventing WAW hazards in pipelined processors are *register scoreboarding* [59], and *register renaming* [60]. The DEC Alpha 21164 [16] uses *register scoreboarding.* This is illustrated in Figure 1.2 [1] . Within the machine's pipeline is a scoreboard for representing register state. When an instruction is issued, its destination register is marked pending, and cleared when the instruction completes. When a subsequent instruction that needs to write or read from a pending register is at the issue stage, the pipeline stalls until the register is written. No instructions proceed past the pipeline stage containing the scoreboard

[1] The instruction set used in the examples is the MAP instruction set. Most operations take three arguments. The first two are operands, the third is a destination. More details on this instruction set can be found in Appendix A.

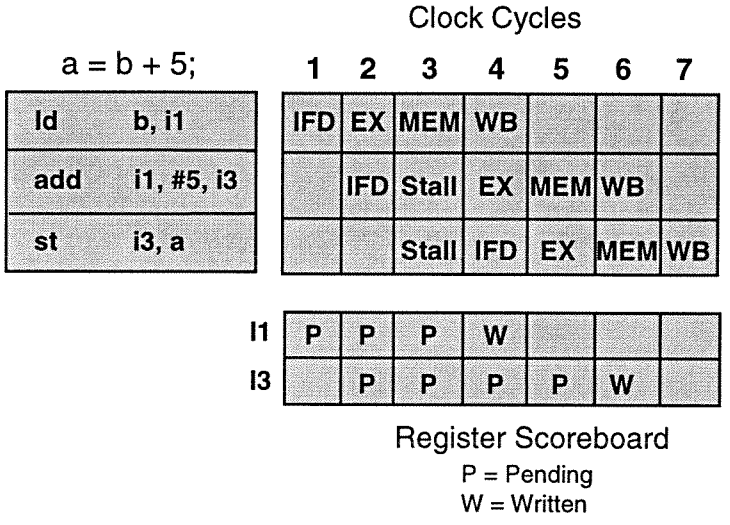


Figure 1.2: **Register Scoreboarding**. This figure shows a simplified machine pipeline with four stages, Instruction Fetch and Decode (IFD), Execute (EX), Memory Access (MEM) and Writeback (WB). The sample code is shown progressing through this pipeline. Due to the data dependencies in the code, the `add` is not able to proceed to the EX stage until the `ld` enters the WB stage. The *stall* of the `add` delays the entrance of the `st` to the IFD stage. The state of the register scoreboard is also shown. When an operation enters the WB phase, the `Pending` state of the register is cleared in the first half of the cycle, allowing the dependent instruction to enter the EX phase. In this situation, the needed value is *bypassed* directly into a functional unit input, as well as being written to the register file.

until the condition which caused the stall has been handled. This form of scoreboarding eliminates both RAW and WAW hazards, but prevents an opportunity of dynamically eliminating "false" WAW hazards: those hazards introduced as a result of how the program is executed.

Register renaming, such as used in the MIPS R10000 [43], provides the opportunity to eliminate the false WAW hazards by using more machine registers than are visible to the programmer. When an instruction is issued, the register name table is updated so that the destination register is given a new physical name and the virtual to physical translation is recorded. Subsequent instructions that read the virtual register are mapped to the physical register. If another instruction that targets the same virtual register is issued, then a new physical register is allocated and the virtual to physical translation is modified. Subsequent reads to the virtual register name will find the new physical register. Register renaming allows the number of live registers to be larger than the number of registers that can be named by the compiler. However, if the number of live registers is greater than the number of physical registers, the pipeline (like that of the scoreboard solution above) must stall.

Both scoreboarding and register renaming are suitable for today's architectures with single register files and a limited number of functional units. However, both techniques require access to global structures: the scoreboard or the name table. These structures will have latency problems similar to those of a monolithic register file as the number of functional units grows.

In the case of scoreboarding, the latency for accessing a global scoreboard would be incurred more than once for each instruction being issued. When a functional unit is ready to issue an instruction, a signal must be sent to the scoreboard to attempt to mark the target register pending. Execution cannot proceed until a return signal is received indicating that the register is now marked pending awaiting the completion of the instruction. When the instruction completes, the scoreboard needs to be written

to update the state for the target register. For operands, it is necessary to check the state of the register, and delay execution until the source register is known to be full. The checking of the source operands can occur concurrently with the checking of the target register if enough ports on the scoreboard are provided.

For register renaming, it is necessary to examine all instructions that are ready to issue in a given cycle to determine which subset of those instructions can be issued. The logic required for this comparison needs to have write ports from all functional units, and grows in size as a function of the square of the number of these ports. The size and latency issues for this logic would be very similar to those encountered for a monolithic register file.

The software-only solution found in VLIW machines is applicable to machines in which all resources are scheduled in the compiler. The work in this thesis extends that work so that the software cooperates with the mechanisms provided in the hardware to handle the non-determinism present in the systems of today and tomorrow.

The complication posed by use of a partitioned register file occurs when a value needs to be written into a remote register file. One of the following conditions must be met for the transfer to be initiated:

- The writing functional unit must have determined that it is safe to write the remote register
- The remote register file must be capable of delaying the processing of the write until it is safe for it to occur

If all of the processors are executing in lockstep, as in a traditional VLIW, and the latency for memory accesses either is known or has a hard upper bound, the compiler is able to statically ensure the first of these conditions. Otherwise, the first of these conditions can only be met if there is global sharing of information among register banks and functional units. In particular, each functional unit needs to be

able to access the scoreboard associated with all register banks. This requirement for a globally accessible scoreboard has delay problems similar to those of a monolithic register file. In the absence of known memory latency and hardware write interlocks, WAW hazards can occur even within a single cluster of the MAP chip.

The second condition requires additional hardware, a queue, and represents a different organization of the processor. The finite nature of the queue can result in hardware deadlock. In addition, support for the queue can increase compiler complexity as queue write order could be critical.

1.4 A New Way of Organizing a Processor

The Multi-Alu Processor (MAP), the processor for the MIT M-Machine, uses several mechanisms to improve processor throughput relative to a super-scalar or VLIW architecture. These mechanisms include loosely-coupled clusters of processors, and hardware multithreading. The MAP clusters are each similar to a super-scalar processor, containing integer, memory, and floating-point functional units with a common register file. All of the clusters within a single MAP have the ability to write the register files associated with all other clusters. Figure 1.3 shows the aspects of the MAP architecture that are relevant to this thesis.

The MAP processor is a prototype of future systems that will contain multiple ALUs and disjoint register files. Unlike VLIWs, the MAP cannot be completely statically scheduled, as the different clusters are allowed to issue instructions independently and memory latencies are variable due to the cache and virtual memory system. The MAP has nine functional units, three each of floating-point, integer, and address, which are organized into three clusters, each of which contains integer, memory, and floating-point units, as well as an integer and a floating point register file. The clusters communicate and synchronize through registers, as a portion of

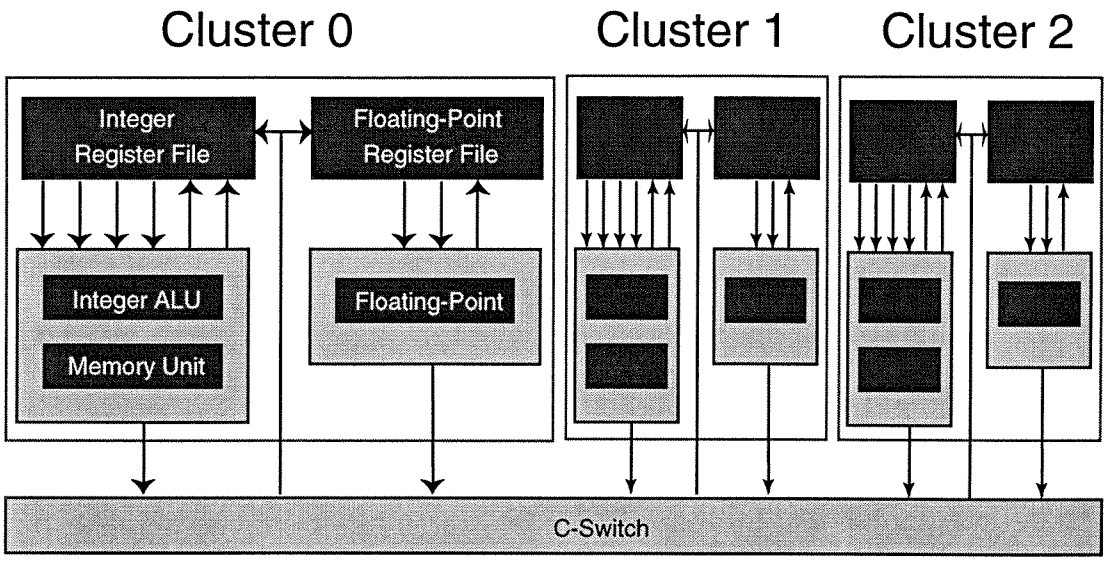


Figure 1.3: **Abstract View of The MAP Processor**. The MAP Processor has three clusters each of which contains an integer ALU, double-precision floating-point unit, and a memory unit, as well as integer and floating-point register files. Data can be transferred register-to-register across the Cluster Switch (C-Switch). The arrows indicate read and write ports, each of which is capable of carrying one machine word in the indicated direction each clock cycle.

each cluster's register file may be written by other clusters.

Register synchronization is performed using a scoreboard and hardware RAW hazard detection. When an instruction issues, its destination register (if the destination is in the same cluster in which the instruction issues) is marked empty. When the instruction completes, its destination register is marked full. The complication occurs when an instruction from one cluster targets the register file of a remote cluster. In this case the register is not marked empty as the hardware required to do this is prohibitively expensive. Instead the receiver is required to execute an `EMPTY` instruction in order to mark the register empty prior to receiving the data. This allows RAW hazards to be eliminated even for remote register writes. The register scoreboard prevents RAW hazards and in-order instruction issue prevents WAR hazards; no hardware is provided to detect WAW hazards.

Further compiler work is required to take advantage of the loosely coupled pro-

cessor clusters and low interaction latency of the MAP chip. Non-determinism in execution timing requires that the compiler choreograph synchronization and communication between clusters. The current approach being explored for this work is scheduling the code within the compiler as if it were a VLIW with all units running in lockstep, and inserting code to synchronize clusters whenever an inter-cluster write is required. A variety of other schemes are also under consideration, including treating each cluster as an independent processor, and using inter-cluster writes as a very efficient form of inter-processor communication. Any technique that is used will require inserting code to synchronize two clusters prior to transferring data between them. Trivial extension of SRS will allow it to be used in the choreography between the clusters. Prior to permitting an inter-cluster register write, the compiler can determine the state of the register to be written using the software scoreboarding described here. If the local register to be written by the remote cluster is not full, the instruction notifying the remote cluster can be delayed by inserting a register barrier on the local cluster. This prevents the remote cluster from writing the local register until it is safe from WAW hazards.

1.5 Related Work

The work presented in this thesis is most closely related to previous work in the fields of register allocation, and instruction-level parallelism (ILP). Previous work in computer architecture for ILP was presented in Sections 1.1 and 1.3. This section presents some of the background material in register allocation, and in compiler support for ILP.

The standard technique for performing register allocation is register coloring. This was originally presented by Chaitin [9, 8]. In the last decade there has been a significant number of publications seeking to improve upon this technique [2, 4, 11, 7, 5, 3].

This is still an area of active research. Recent research has focused on developing techniques to refine graph coloring to reduce the number of spills that are inserted into code. Most recently, work [6, 24] has focused on using different heuristics to decrease the amount of spilling that is performed. One of the fundamental assumptions of register coloring is that instructions will not be reordered once register assignments have been made. The instruction scheduling performed by the Multiflow compiler renders this technique unusable [23].

Compiler techniques for eliminating both RAW and WAW hazards have been proposed and implemented. Significant work on static scheduling for instruction-level parallelism, particularly in scheduling beyond basic block boundaries, was performed at Yale University [17], and was implemented in the Trace [12] family of VLIW computers produced by Multiflow. The Multiflow machine had up to 28 execution units and a register file partitioned across the units. The entire machine executed in lockstep with fully predictable latencies. The linchpin of this technology was the compiler [37] which statically scheduled long traces of code for instruction level parallelism. The compiler performed memory bank disambiguation to prevent multiple references to the same bank of memory in a given cycle, and there was no data cache. As all latencies could be statically determined and no out of order execution was allowed, all RAW and WAW hazards were eliminated by the compiler, and without complex hardware. Further details about the Multiflow compiler and trace-scheduling are presented in Section 1.6.

1.5.1 Instruction Scheduling for ILP

There have been a variety of techniques proposed for increasing the amount of ILP available to the scheduling phase of a compiler. There are two classes of these techniques. One group focuses on scheduling beyond the boundaries of a basic block.

The other group works on loop transformations to increase parallelism within a basic block.

Trace-scheduling [17, 37] is covered in detail in Section 1.6. In summary, this algorithm uses heuristics to predict which direction conditional branches will go at runtime. This information is used to coalesce basic blocks into a single unit for scheduling called a trace. This algorithm performs similar code motion to that used for reordering instructions within a basic block to increase ILP, but allows code to move past conditional branches under some circumstances. The conditions under which such code motion is allowed are expanded upon in Trace Scheduling-2 [21].

Superblocks [33] are a variation on traces. The key difference is that a superblock only has one entrance point at the top, whereas a trace can also have entrance points into the middle. A hyperblock [38] is similar to a superblock, but contains predicated instructions. That is, a hyperblock may contain code from mutually exclusive basic blocks, with predicates on the execution of some instructions controlling whether or not they are executed. The target machine for such a construct supports predicated instructions in hardware.

There have been a variety of compile-time techniques used for increasing the amount of available ILP within a loop. In particular, software pipelining [36] exposes parallelism by overlapping instructions from different loop iterations without violating loop-carried data dependencies. Similarly, loop unrolling, duplicates the code within a loop which has the same overlapping benefits of software pipelining, and also reduces the number of conditional branches taken during loop execution.

Branch prediction can also be very powerful for increasing ILP. The compiler statically predicts which way a branch will go at run time, and schedules the code using that assumption. This can result in an increase in code size if an alternative piece of code is generated for when the branch takes the other direction. Some of

this code bloat can be reduced if the target architecture has predicated operations. There has also been work done on the feasibility of performing compiler-time branch prediction based on information gathered during previous runs of the program [22].

Another important area of research relating to compiler scheduling for parallelism is that of inter-procedural analysis. Traditionally compilers, including the Multiflow compiler, have restricted their data analysis to within individual procedures. Recently work has been conducted into the possibility of extending traditional analyses across procedural boundaries [61, 29, 28]. This work has the possibility of increasing the amount of ILP made available to the scheduling phase of a compiler.

1.6 The Trace Scheduling Algorithm

The motivation for trace scheduling is found in the problem of compiling for ILP. In general, it is difficult for a compiler to locate enough parallelism within a basic block to sustain utilization of multiple functional units. A compiler using the trace-scheduling algorithm seeks to overcome this obstacle by performing scheduling on a larger unit than a basic block. This larger unit is called a trace. A trace is allowed to span multiple basic blocks, and may contain conditional branches within it. Traces are not allowed to contain loop back edges. One important difference between a basic block and a trace is that a trace is allowed to have multiple exit and entry points.

Figure 1.4 shows the steps performed by a traditional basic-block compiler. Within such a compiler the ordering in which blocks are scheduled is based on their position within the control-flow graph. This graph is traversed in some ordering, generally breadth- or depth- first. In contrast, trace-scheduling uses heuristics to predict which paths through the control-flow graph are most likely to be followed, and assembles these basic blocks into a trace to be scheduled as a unit. Figure 1.5 shows the selection of a trace from a control flow graph. The goal of the trace-picking

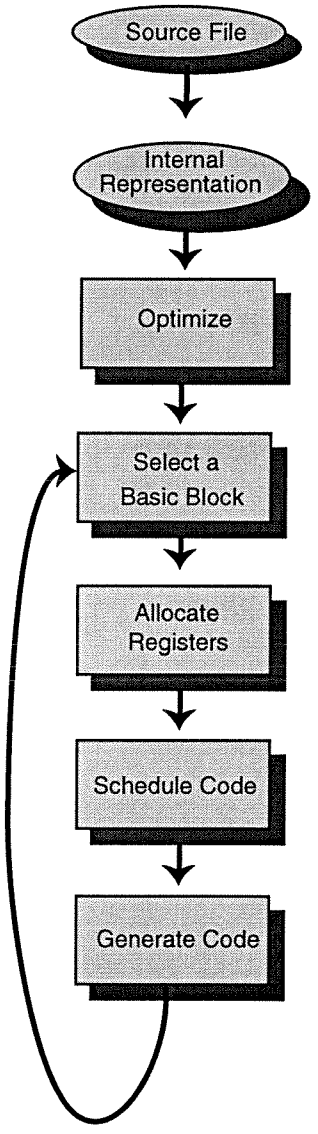


Figure 1.4: **Operation of a Basic Block Compiler.** A traditional compiler iterates over basic blocks in the program graph. For each block it first allocates registers, and then schedules operations onto the functional units of the target machine. When it has scheduled all of the basic blocks, it terminates.

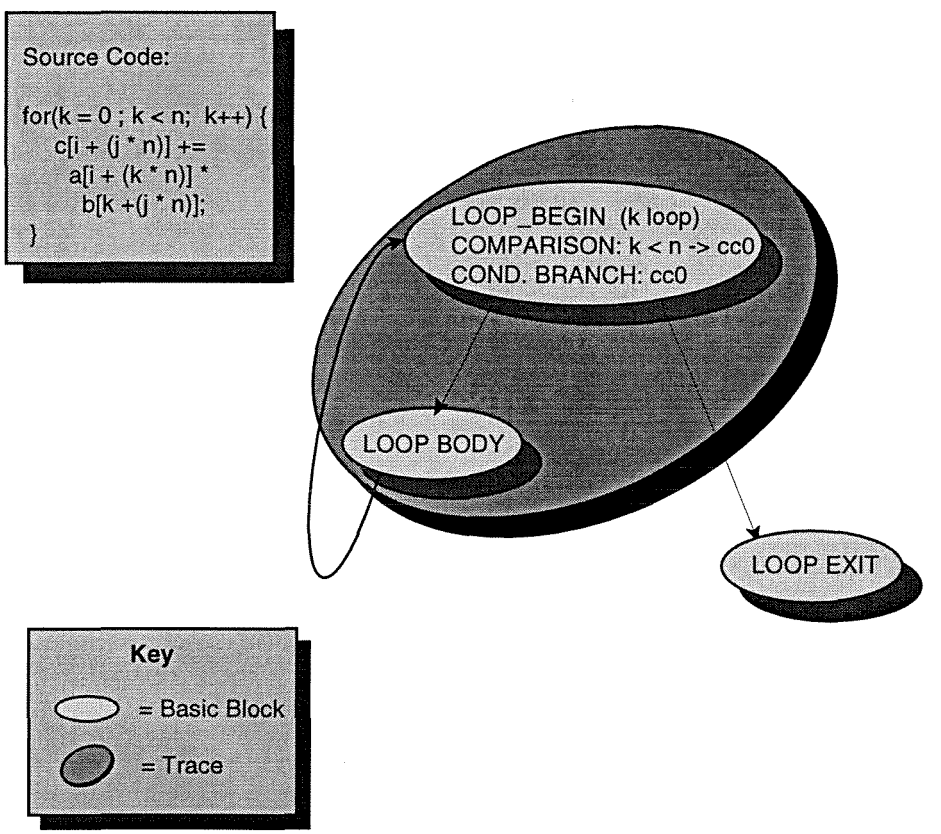


Figure 1.5: **Division of a Program into Basic Blocks and Traces.** The Trace-picking heuristics predicts which direction the branch ending a basic block is most likely to go. The code most likely to be executed is added to the trace containing the branch. For loops, the heuristics assume that the loop body is more likely to be executed than the loop exit code.

heuristics is to identify the performance critical portions of the code first, so that they are not bound by resource constraints imposed by less important portions of the program. In particular, the trace-picking heuristics are intended to favor code composing inner loops of scientific computations. The general flow of operations

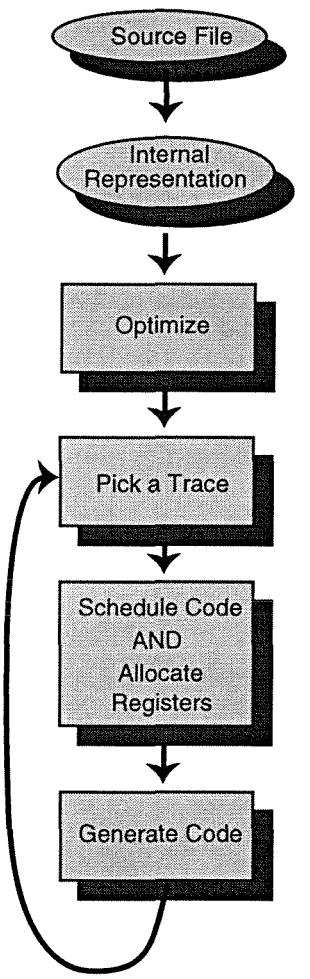


Figure 1.6: **Operation of the Multiflow Compiler**. The Multiflow compiler picks traces out of the program graph, and performs a single scheduling step which performs both register allocation and functional unit assignment at the same time. When the entire program has been scheduled, the compiler terminates.

within the Multiflow compiler is shown in Figure 1.6.

This compilation strategy involves significant departures from traditional compilers. These include performing register allocation and instruction scheduling as part

of the same compilation phase; reordering instructions to shorten a schedule; and duplicating instructions at trace split and join points. One of the most obvious of these departures is the division of the program into traces described above. The following sections provide details of some of the other interesting aspects of trace-scheduling.

1.6.1 The Phase Ordering Problem

In a traditional basic block compiler there are usually independent scheduling phases for register allocation, and instruction scheduling. The separation of these two processes creates a phase ordering problem [23]. If register allocation is performed first, dependencies on register usage which are unrelated to program data dependencies can be introduced into the schedule. If instruction scheduling is performed first, it can be impossible to perform register allocation for the resultant schedule. To compensate for these problems, it is necessary to insert spills and restores after much of the scheduling information has been destroyed. In general, traditional compilers will perform register allocation first, as this results in less inefficiency.

The Multiflow compiler uses a different approach. Register allocation and instruction scheduling are performed as part of the same phase. The emphasis on this process is that priority for registers should go to the instructions that are going to be issued most frequently. One of the ramifications of this decision is that register coloring [9, 8] can not be used during trace-scheduling. One of the assumptions of register coloring is that instructions will not move relative to each other. As described in Section 1.6.2 the reordering of instructions is important to the efficiency of the code generated by the Multiflow compiler.

The key idea used for register allocation in the Multiflow compiler is *delayed binding*. Values are not assigned to registers until they are actually referenced. When

the first trace in the program is selected it has no constraints on register usage. Recall that the trace-picking heuristics are intended to select the performance-critical inner loops of scientific computations to be scheduled prior to any other portions of the code. If the target machine has sufficient registers, the inner loops will be executed with no spilling or extraneous copies. Any necessary spills are pushed out of the critical path of the program.

This mechanism functions by attaching information about bindings between variables and registers to scheduled traces. This process is called *bookkeeping* within

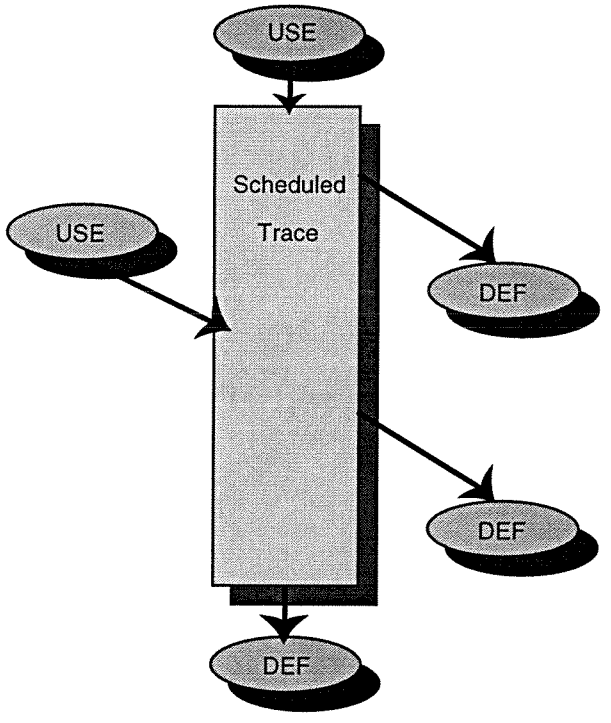


Figure 1.7: **Encapsulation of Delayed-Binding Information**. Once a trace has been scheduled, variable binding information is inserted into the program control-flow graph. A `USE` node indicates where this trace assumes that variables are upon entry. A `DEF` node indicates where variables are left upon exit.

the Multiflow compiler literature [17, 37]. This information is in the form of special nodes inserted into the control-flow graph. These nodes are inserted in between a piece of code which has just been scheduled, and any unscheduled code. `USE` nodes

are placed at *join* edges, and contain information as to where the scheduled code assumed variables to be located. When the adjoining unscheduled code is scheduled, it must ensure that these variables are left in the proper place. `DEF` nodes are placed at *split* edges, and contain information as to where the scheduled code has left variables. When the adjoining unscheduled code is scheduled, it must use these nodes to determine the initial location of the variables. This is shown in Figure 1.7. Until a variable is actually referenced, the location it is bound to is a temporary. Section 1.6.3 contains further discussion of *split* and *join* edges.

The generic name for these nodes is a *Value-Location Mapping* (VLM) [23]. When a trace is scheduled it has to reconcile the information in its entry and exit VLMs. If it is not possible to maintain a desired register binding throughout an entire trace, and no other registers are available for temporary storage, the variable in question must be spilled to and restored from memory.

1.6.2 Instruction Reordering

Instruction reordering is another technique that increases the amount of available ILP in code output by a compiler. The idea behind this reordering is that the sequential ordering of instructions within a program does not necessarily represent the data dependencies of instructions. If two operations do not share any data dependencies, then their execution order can be reversed without affecting the correctness of the program. Some instructions can be moved past conditional branches. When this occurs, it is sometimes necessary to insert *compensation code* to ensure that all of the required instructions are executed whichever way the conditional branches go. This section deals with code motion that does not require *compensation code.* Section 1.6.3 describes *compensation code*, and the circumstances under which it is required.

The data dependencies between instructions are represented as edges in the DAG of a trace. An instruction is unavailable for scheduling until all instructions on which it depends have been scheduled. Once all of these constraining operations have been scheduled, the instruction is *data ready*. Any ordering in which *data ready* instructions are scheduled should be safe, as these instructions are guaranteed to not have any true data dependencies. It is still necessary to ensure that no false data dependencies are introduced through register reuse. Figure 1.8 shows an example of legal code

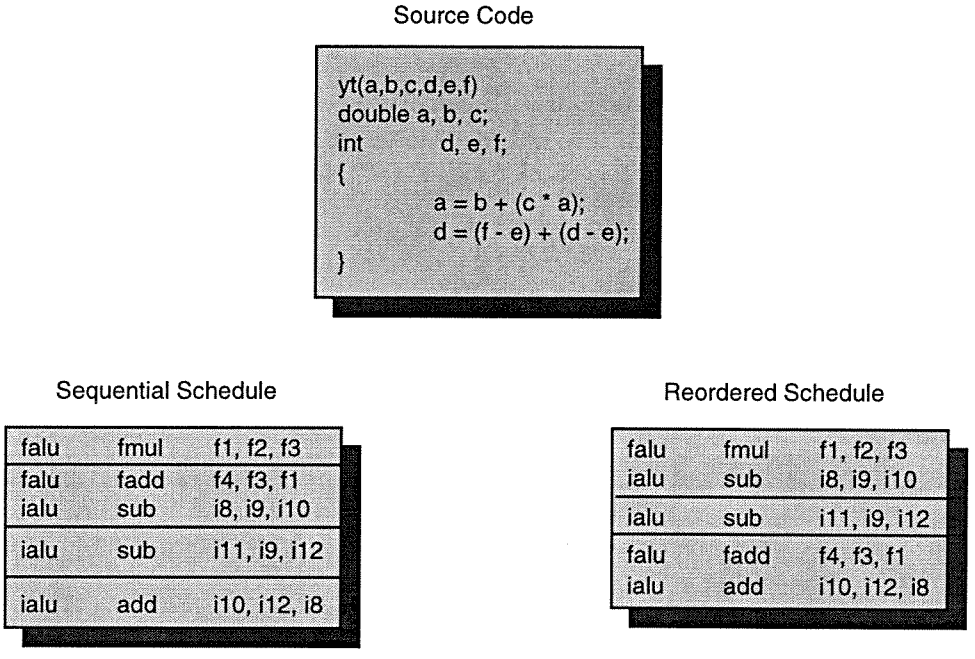


Figure 1.8: **Code Motion to Increase ILP**. If this code is scheduled based purely on the order in which operations occur in the source program, four cycles are needed to issue all of the instructions. Since there would certainly be a floating-point pipeline stall between the first and second instruction, this would take several additional cycles to complete. If the code is reordered, it is possible to increase the distance between the two floating point operations, thereby decreasing the duration of the pipeline stall. Also, the reordered schedule requires one fewer cycle to issue.

motion.

It is legal to move an instruction above a split if the result of the split is not to memory, and is not live on the off-trace edge. That is, if the result of the operation might be accessed if the branch goes opposite to the predicted direction, it cannot

legally be moved. This code motion requires that the hardware be capable of delaying or suppressing the exceptions that might result from issuing an instruction out of order. For example, moving a division above the safety check ensuring that the divisor is not zero can cause a divide-by-zero exception.

This processing is similar to the scheduling performed by a compiler for a pipelined machine in order to minimize pipeline stalls [53, 25, 31]. The ability to schedule across basic block boundaries significantly expands the possibilities for such code motion. For a pipelined machine, the emphasis is on trying to fill all of the issue slots between the start of an instruction, and the clock cycle when it is legal to access the result of the instruction. For superscalar and VLIW machines, there is the added goal of increasing the available ILP in the program to take advantage of the available functional units.

1.6.3 Management of Splits and Joins

When managing conditional branches, special handling is necessary to ensure that any instructions that have been moved beyond a branch are executed on all appropriate paths through the program; and that the machine resource restrictions are adhered to whichever direction of the branch is taken. The Multiflow compiler models these changes in control flow as *splits* and *joins.* Within the program graph, these transitions are modeled as edges which can contain instructions, and information about both machine state and the binding of variables to registers. The instructions within such an edge are *compensation code* required to maintain program correctness. The information about machine state is contained within *partial schedules.* Section 1.6.1 describes the management of variable-register bindings. Section 1.6.2 describes code motion beyond *splits* and *joins* that do not require *compensation code.* This section discusses *compensation code* and *partial schedules.*

Compensation Code

When an instruction is moved below a *split* or above a *join*, it is necessary to insert *compensation code* to ensure correctness of the compiled code. An example of the insertion of *compensation code* is shown in Figure 1.9. Any instruction that has

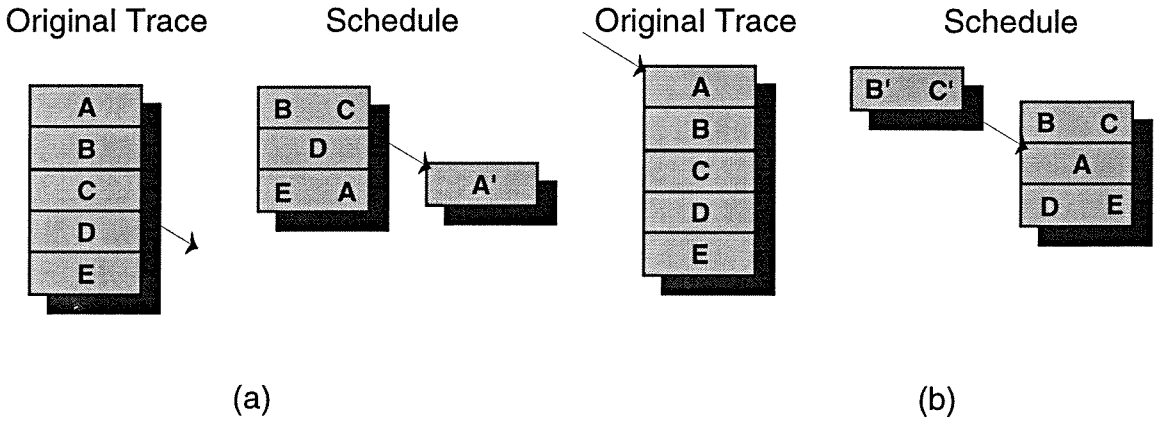


Figure 1.9: **Code Motion Below Splits and Above Joins.** **(a)** The original trace contains a *split* after instruction `C`. When the trace is scheduled, `A` is moved below the split. A copy of `A`, denoted `A'`, is added as *compensation code* on the outgoing edge. **(b)** The original trace contains a *join* before instruction `A`. When the trace is scheduled, both `B` and `C` are moved above the *join*. Copies of `B` and `C`, denoted `B'` and `C'`, are added to the incoming edge as *compensation code*.

moved beyond a branch will have to occur in at least two places in the generated code. This duplication of instructions can lead to code explosion. The Multiflow compiler contains safety mechanisms to restrict this code duplication. In particular, if a program is deemed to have grown too large, all further code motion which would require *compensation code* is forbidden. As this prohibition is unlikely to be invoked until after the traces most likely to be frequently executed have been scheduled, it should have little effect on the overall efficiency of the generated program.

Instructions that occur before a *join* point can tie up hardware resources. If one instruction is followed by stall cycles inserted into the schedule, and a *join* is allowed to happen once this instruction has issued, it is more efficient to wait until the stalls have happened as well. This is shown in Figure 1.10.

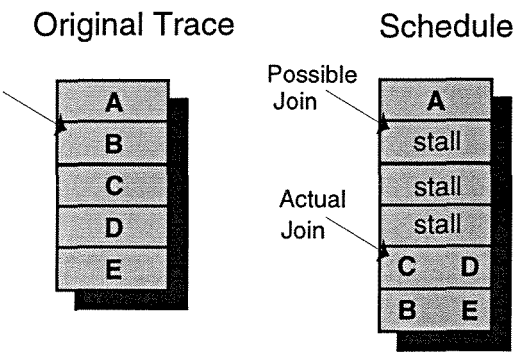


Figure 1.10: **Selection of Join Point to Minimize Hardware Delay**. The original trace contains a *join* after instruction `A`. When `A` is scheduled, it is followed by three hardware stall cycles. While the *join* could happen during the first of these stalls, it is more efficient to delay the *join* until just after the last stall.

In order to balance the possible delays due to hardware constraints with the amount of *compensation code* that is generated, the compiler selects the cycle at which a *join* occurs. Joining as early as possible minimizes *compensation code*, but can result in delays.

Partial Schedules

When transition between traces occur, it is possible that pipelines or other machine resources might be in use within an already scheduled trace. It is necessary to ensure proper modeling of this hardware resource usage. If there is a hardware resource such as a floating-point divide unit which can accept a new instruction once every five cycles, and an instruction using this resource is issued three cycles before a *split* point, this resource usage must be modeled to prevent a conflict when an adjacent trace is joined. This is represented as a *partial schedule*. The usage of the resource is copied onto the edge, but the instruction that requires the usage is not. This is shown in Figure 1.11. The resource information is added on to the top of the trace being joined. The scheduler ensures that this resource is not over-subscribed. An example of the importance of *partial schedules* can be found in Section 4.1.

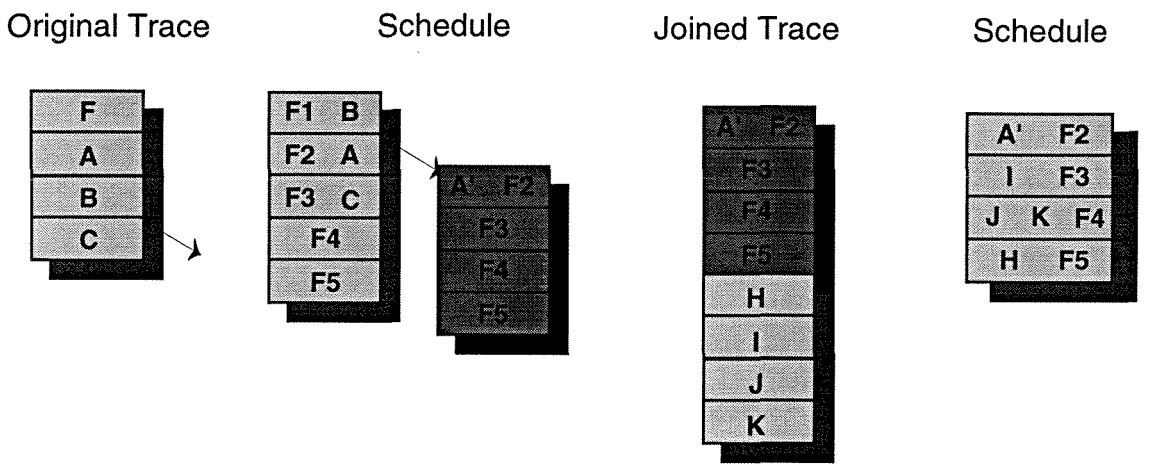


Figure 1.11: **Resource Usage Represented as a Partial Schedule.** Instruction F is a floating-point instruction that ties up some hardware resource for five cycles. If a split occurs before those five cycles are up, the resource usage information must be copied onto the off-trace edge. When the trace being joined by this *partial schedule* is scheduled, both the *compensation code* and *partial schedule* information are incorporated into the trace. During scheduling, operations from lower down in the trace are allowed to move up and overlap with the resource usage information as long as they don't need the busy resource.

A busy resource does not consume an instruction issue slot. A machine could have one floating-point issue slot every cycle; and floating-point divide and add could use different hardware resources. The busy status of the divide unit would not prevent adds from being issued.

1.7 Overview of the Dissertation

The current trend in high-performance microprocessors is towards improving performance by increasing the number of functional units on a single chip. These chips apply complex hardware solutions to the problem of register synchronization. The register file size, and accompanying wire lengths and delays, significantly increases as the number of functional units go up. The most obvious solution to this problem, partitioning the register file, rules out hardware implementations of existing register synchronization schemes. The delays incurred by the monolithic register file are still

present in the synchronization scheme for a partitioned register file. The limiting effect on processor clock speed caused by the delays entailed in a large number of functional units dictate that register files be partitioned, but the partitioned register file system requires a new approach to register synchronization.

This dissertation presents a novel compiler-based approach to this problem. This solution, the Software Register Synchronization (SRS) algorithm, is a synthesis of a known hardware technique, *register scoreboarding*, with an existing compiler scheduling algorithm, *trace scheduling*. The SRS algorithm relies on the existence of a local scoreboard at each register file which enforces a pipeline interlock when attempting to read a register which is marked as empty. This operation can be managed using purely local information. The Multiflow trace-scheduling compiler provides the ability to make decisions about register selection at the same time as functional unit selection. This allows the SRS algorithm to bias register use away from empty registers, as well as ensuring that local synchronization is performed on any potentially empty register prior to overwriting the register. The trace scheduler also provides a framework for propagating information between traces at compile time. This framework is utilized to propagate register state information across trace boundaries. Finally, the Multiflow compiler provides static scheduling and instruction reordering to increase the amount of available Instruction-Level Parallelism (ILP) in a program.

1.8 Contributions of this Dissertation

This dissertation makes the following contributions:

- It identifies register size expansion as a limiting factor on degree of ILP in processors, proposes partitioning of register files to circumvent this limitation, and

describes the problems in implementing traditional hardware register synchronization techniques in the presence of a partitioned register file.

- It proposes and demonstrates the correctness of a compiler algorithm, Software Register Synchronization (SRS) to manage register synchronization in both conventional super-scalar designs, and super-scalar designs with partitioned register files.
- It evaluates the performance characteristics of code compiled using the algorithm, and evaluates the M-Machine architecture as a compiler target.

1.9 Organization of the Dissertation

Chapter 2 describes a compiler algorithm for managing register synchronization within a system with a monolithic register file. This includes a description of the register synchronization problem, and a demonstration of the correctness of the algorithm. Chapter 3 extends the algorithm to function within a trace-scheduling compiler. This description includes details of the implementation of this algorithm within the Multiflow compiler, and necessary changes to the demonstration of correctness. Chapter 4 extends the algorithm to a multi-cluster environment. This includes the additional implementation notes, as well as changes to the demonstration of correctness. Chapter 5 describes some implementation details of the retargeting of the Multiflow Compiler to the M-Machine, accompanied by a critique of the M-Machine instruction set architecture. Chapter 6 provides experimental results. These results include an evaluation of the single cluster algorithm relative to hardware solutions for register synchronization. Chapter 7 proposes future directions for this work, and presents conclusions.

Chapter 2 Register Synchronization for a Monolithic Register File

This chapter reformulates the issue of register synchronization, and presents a compiler algorithm for ensuring correct execution of compiled code on a processor with a monolithic register file. The original version of the algorithm is for a traditional basic-block compiler targeted for a single-cluster of the MAP. Following chapters will introduce additional constraints, and discuss the changes necessary to the algorithm to manage these additional issues. For the purposes of this discussion, the target architecture is defined as only enforcing register synchronization on reads. A demonstration of the correctness of the algorithm is presented.

Existing processors demonstrate that register synchronization is readily manageable for a monolithic register file using either *register scoreboarding* [59], or *register renaming* [60]. The algorithm is presented in this form to provide a base case of compatibility with existing techniques.

The set of possible machine operations can be divided into three categories based on their handling of register synchronization state. `READ` operations are the only operations that perform synchronization.

- `LOAD` places a value into a *destination* register. `LOAD`s can complete asynchronously. When a `LOAD` is initiated a synchronization flag associated with the *destination* register is set. This flag is cleared when the `LOAD` completes.
- `READ` takes a value from a *source* register and uses it to perform some operation (such as *add, subtract, compare, etc.*). `READS` will not begin until the

synchronization flag on the *source* register is cleared.

- WRITE places a value into a *destination* register. WRITES will complete even if the synchronization flag on the *destination* register was set when they started. After a WRITE has completed, the synchronization flag on the *destination* register is always cleared.

2.1 Example 1: Delayed Loads

Figure 2.1 demonstrates the need for register synchronization: as WRITEs do not respect the state of their *destination* registers, out-of-order instruction completion can result in incorrect program execution. The value **p* is LOADED into *r1*. If $q = 0$, the program will stall until the LOAD completes, and then READ *r1* and add it to *r2*. If $q \neq 0$, *r1* could be WRITTEN before the LOAD completes. Figure 2.2 shows a worst-case timing sequence. In this case, the test fails, and *#1* is WRITTEN to *r1*. Prior to the return, the LOAD completes, placing **p* in *r1*. The incorrect result is returned to the caller.

```
a = *p;          LOAD    p, r1        ; LOAD from location p into r1
if(q == 0)       EQUAL   q, 0, cc0    ; Check (q == 0)
                 BF      cc0, L1      ; If false, branch to L1
   a = a + b;    ADD     r2, r1, r2   ;Add r1 to r2
                 JMP     L2
else             L1:
   a = 1;        MOVE    #1, r1       ; WRITE r1
                 L2:
return a;        RETURN  r1
```

Figure 2.1: **Delayed Load**. This shows the source code and assembly language for a code fragment containing a WAW hazard. If the FALSE direction of the branch is taken, there would be two consecutive writes to the same register.

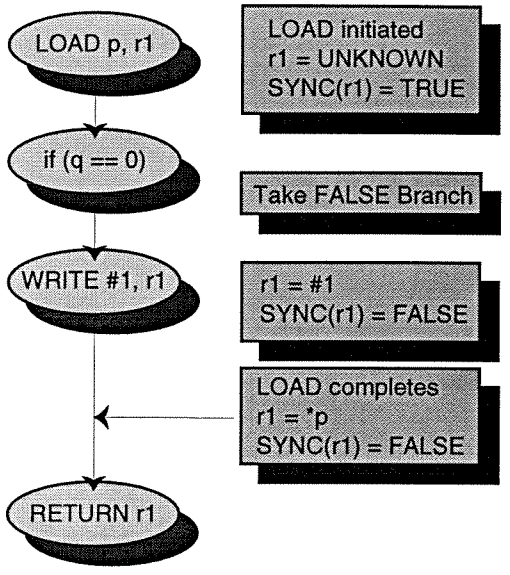


Figure 2.2: **Worst-case Timing for Delayed Load**. This diagram shows the timing of operations for the Delayed Load example that would generate the incorrect answer. In between the initiation and completion of the LOAD operation which uses r1 as its target, the number 1 is written into the r1. At the end of this execution r1 is supposed to contain 1, but contains instead *p.

2.2 Example 2: Multiple Loads

Figure 2.3 shows a second situation in which register synchronization is necessary: Two consecutive LOADs to the same register can complete in the wrong order, leaving the incorrect value in the register. The value *p* is LOADED into *r1*. If $q = 0$, the value *b* is LOADED into *r1*. This creates a race condition to determine the contents of *r1*. Figure 2.4 shows a worst-case timing sequence. In this case in between *p* is LOADED into *r1*. As $q = 0$, *b* is LOADED into *r1*. The LOAD of *b* completes immediately. Then the LOAD of *p* completes. The incorrect result is returned to the caller.

```
a = *p;          LOAD     p, r1        ; LOAD from location p into r1
if(q == 0)       EQUAL    q, 0, cc0    ; Check (q == 0)
                 BF       cc0, L1      ; If false, branch to L1
    a = *b;      LOAD     b,r1
                 L1:
a = a + c;       ADD      r1,r4,r4     ; READ r1, r4; WRITE r4
return a;        RETURN   r4
```

Figure 2.3: **Multiple Loads**. This program contains the source code and assembly language for a more subtle register synchronization problem. If the first load results in a remote memory access, and the second load hits in the cache, the second load could complete first, yielding the incorrect answer.

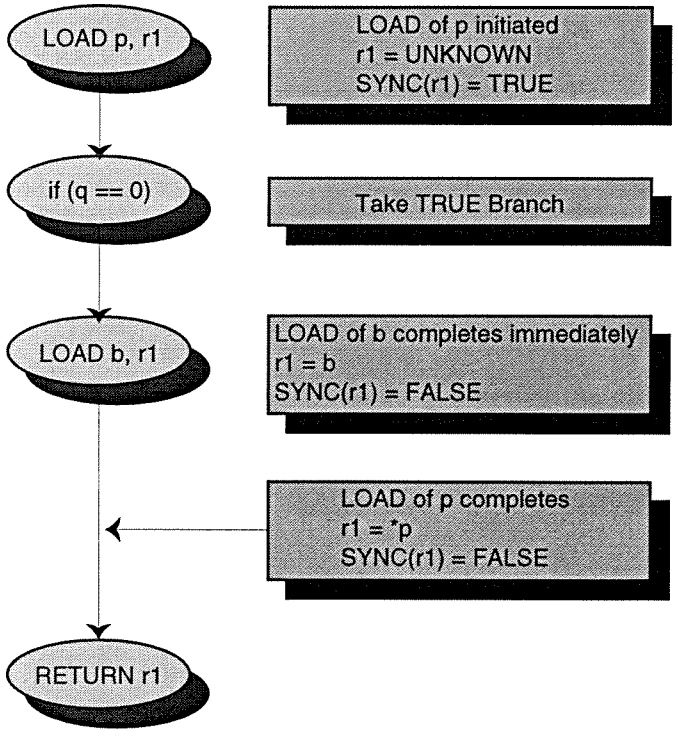


Figure 2.4: **Worst-case Timing for Multiple Loads**. This diagram shows the timing of operations for the Multiple Loads example that would generate the incorrect answer. The first `LOAD` misses in the primary cache, and takes several cycles to complete. The second `LOAD` hits in the primary cache, and completes immediately. The target register, r1, which is supposed to contain *b, ends up containing *p

2.3 Development of Register Synchronization Definition

This section further refines the specification of the register synchronization algorithm. An initial statement of the specification is:

> Guarantee that no `WRITE` or `LOAD` has as its *destination* a register whose synchronization flag is set.

This can be achieved within a basic block using a single forward pass over the block, inserting correction code when a problematic instruction is encountered. To achieve correctness `READS` can be inserted to force synchronization. Complications arise when dealing with transitions between basic blocks, as this requires propagating state information across block boundaries.

To preserve generality, the scheduling of basic blocks is assumed to have no fixed ordering. This is illustrated in Figure 2.5, which shows four basic blocks. Two of these basic blocks have already been scheduled; two of the blocks are as yet unscheduled. One of the unscheduled blocks has been selected to be scheduled. There are operations in both the preceding and succeeding scheduled block which might require action within the current block. If the first instruction in the selected block is `WRITE R4`, it is necessary to first insert a `READ R4`. If the last instruction is a `LOAD R3`, the `READ R3` in the predecessor makes this a safe operation to perform.

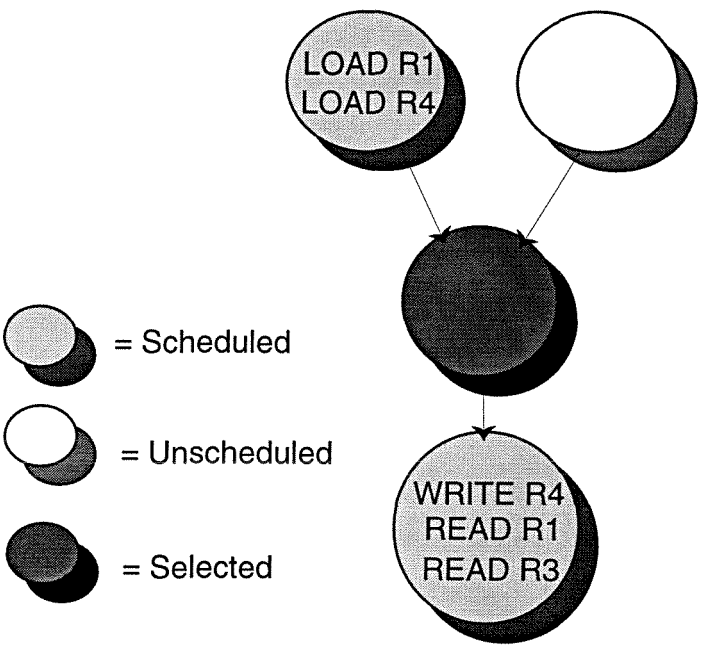


Figure 2.5: **Adjacent Block Scheduling**. During code scheduling, traces are not processed in any fixed ordering. When scheduling a given trace, there could be both predecessors and successors containing operations that can affect the current trace. This diagram shows a portion of a program graph. The trace that has just been selected for scheduling inherits two registers from a scheduled predecessor that are `PENDING` as they are the targets of `LOAD` operations within that predecessor; but sees two registers that are `GROUNDED` by `READ` operations in a scheduled successor.

The problem of ensuring program correctness can be specified as:

> Obtain register state information from scheduled predecessors. Generate a schedule for the current block, inserting synchronizing READS where required. Examine scheduled successors, insert any READS required for transition from current block into scheduled blocks. Ensure that state information made available to all predecessors and successors is correct.

2.4 Overview of Solution

There are three parts to the solution of this problem: scheduling the code for a basic block; maintaining state information to be propagated to unscheduled adjoining blocks; and obtaining state information from scheduled adjoining blocks. The processing can be divided into three phases: entering a basic block, processing a basic block, and leaving a basic block.

There are three possible states for a given register. The state of a register can be propagated downward from a scheduled predecessor, established in the current block, or upwardly-exposed from a scheduled successor. The three states are:

- PENDING. The synchronization bit for the register is set; or the exposed reference to this register is a LOAD.
- FULL. The synchronization bit for the register is cleared. There either is no exposed reference, or the exposed reference is a WRITE.
- GROUNDED. There is an upwardly exposed READ of this register.

2.4.1 Entering a Basic Block

Upon entry to a basic block it is assumed that all registers are FULL. All predecessor blocks are examined. The final register states from any scheduled predecessors are

combined. If a register is PENDING in any scheduled predecessor, it must be treated as if it is PENDING in all scheduled predecessors. As GROUNDED applies only to upwardly-exposed state it is not relevant here.

2.4.2 Processing a Basic Block

For each instruction within a basic block, several stages of processing are performed. All instruction source operands that are in registers are identified, and those registers are marked FULL. The state of the destination register, if any, is checked. If the destination register is PENDING, a READ on the destination register is inserted prior to the current instruction. If the instruction is a WRITE the destination register is marked FULL. If the instruction is a LOAD the destination register is marked PENDING.

2.4.3 Leaving a Basic Block

The final register states from the current block must be consistent with the first usage in any scheduled successors. Within the current block, each register is identified as either PENDING or FULL. If a register is FULL on exit from the current block, than any initial operation using that register in the next block is legal. If a register is PENDING, the only initial operation that is legal is a READ. If the upwards-exposed reference to a register is a READ, the register state is GROUNDED. If the STATE for a register in all scheduled successors is GROUNDED, than the register can be PENDING on exit from the current block. Otherwise, it is necessary to GROUND the register by performing a READ on it prior to exiting the current block. The full set of state transitions is shown in Figure 2.6.

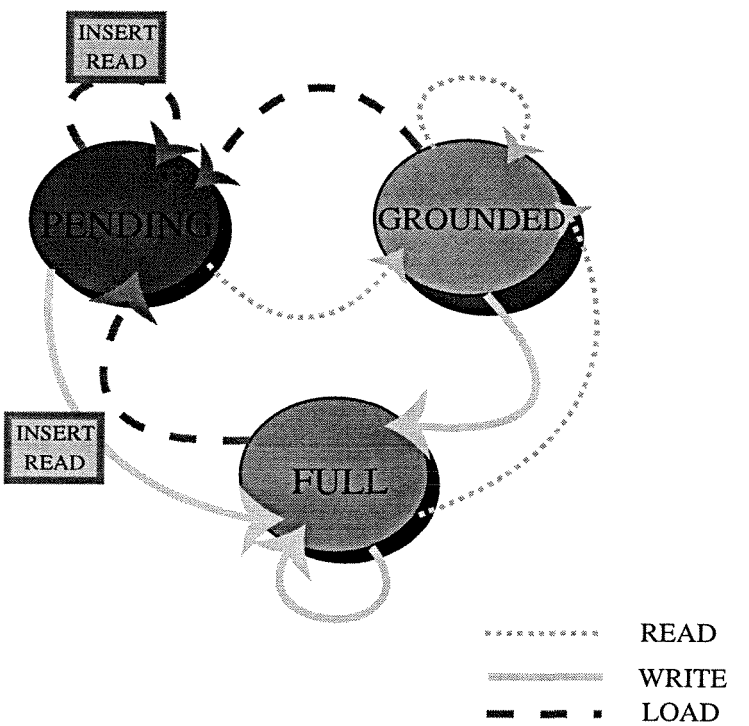


Figure 2.6: **State Transition Diagram.** There are three states which a register can be in. There are three types of instructions that can alter a register's state. This state transition diagram represents the register states as nodes, and the instruction types as edges. The two edges which require the insertion of a synchronizing `READ` to preserve program correctness are so designated.

2.5 Algorithm

Recall that the specification being implemented is:

> Obtain register state information from scheduled predecessors. Generate a schedule for the current block, inserting synchronizing READS where required. Examine scheduled successors, insert any READS required for transition from current block into scheduled blocks. Ensure that state information made available to all predecessors and successors is correct.

As described in Section 2.4 registers can be in one of three states: PENDING registers have their synchronization bit set; FULL registers had their synchronization bit cleared by a WRITE; GROUNDED registers had their synchronization bit cleared by a READ. The GROUNDED state is only relevant for managing transitions from a block to a scheduled successor. Within a block GROUNDED is identical to FULL.

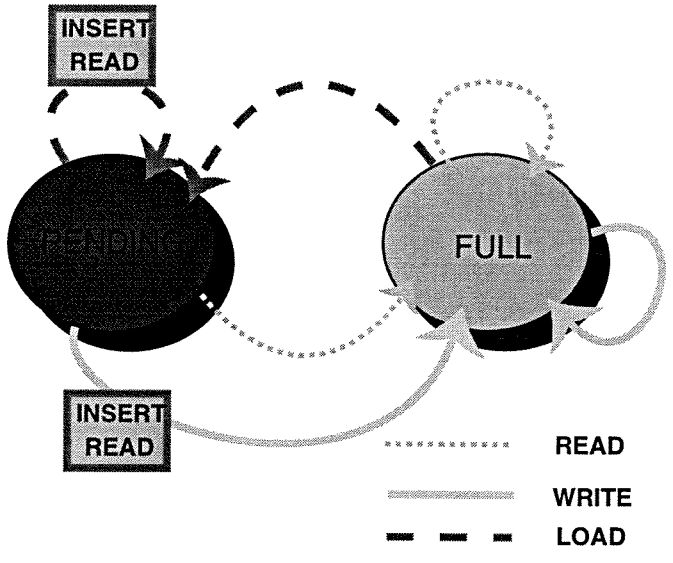


Figure 2.7: **State Transitions for Single-Instruction Scheduling.** While working within a basic block, only two register states need be considered. The third state, GROUNDED is only relevant while reconciling the current block with upwardly-exposed references in a scheduled successor.

Within a basic block register synchronization states are represented as a set, PENDING, of registers with their synchronization bit set. The possible state tran-

sitions for each register referenced in an instruction are shown in Figure 2.7. For any given register, if the register is not in PENDING initially, any operation can be performed without special handling. If the performed operation is a LOAD, the register is placed in PENDING. If the register is initially in PENDING, the only operation which can be performed without special handling is a synchronizing READ. Prior to issuing any other instruction a READ must be inserted to ensure that the register is FULL before it is overwritten. If either a READ or a WRITE is performed, the register is removed from the set PENDING.

Register synchronization information is propagated between basic blocks in two different forms. Information from a scheduled predecessor is in the set PENDING described above. Information from a scheduled successor is in a set STATE which encodes information about the first usage of each register. The possible values for a register in STATE are FULL, PENDING and GROUNDED depending on the first reference to the register in the scheduled successor.

The general outline of the algorithm is shown in Figure 2.8. Iterating until all basic blocks have been scheduled, select a basic block. The routine SelectBlock() chooses the next block to be scheduled. All that is specified about SelectBlock() is that it will select an unscheduled block. Calculate the initial value for $\texttt{PENDING}_b$ for the selected block. Schedule the instructions within this basic block. Calculate $\texttt{STATE}_I$ for this block. This is derived from $\texttt{STATE}_s$ for all scheduled successors of the current block. For any register which was not referenced in this block, set $\texttt{STATE}_b(r)$ for that register to $\texttt{STATE}_I(r)$. Generate any compensation code needed to reconcile $\texttt{PENDING}_b$ with $\texttt{STATE}_I$.

Figure 2.9 shows the algorithm for scheduling a basic block. Iterating over all instructions within the basic block, ensure that all operations are scheduled and all necessary READS are inserted. Determine the value of $\texttt{STATE}_b(\texttt{r})$ for all registers refer-

```
while (not all blocks scheduled)
    b = SelectBlock();
    PENDING_b = ∅;
    for all predecessors p of b
        if (predecessor p scheduled)
            PENDING_b = PENDING_b ∪ PENDING_p;
    for all registers r
        STATE_b(r) = NULL

    Schedule(b);
    STATE_I = JoinSuccessorStates(b);

    for all registers r
        if (r ∈ PENDING_b AND (STATE_I(r) ≠ GROUNDED)
            issue read of r
            remove r from PENDING_b
            if (STATE_b(r) = NULL)
                STATE_b(r) = GROUNDED;
        if (STATE_b(r) = NULL)
            STATE_b(r) = STATE_I(r);
    Mark b as scheduled
```

Figure 2.8: **General Outline of Algorithm.**

enced in the block, and maintain the membership information for the set $\texttt{PENDING}_b$.

The routine `JoinSuccessorStates()` is shown in Figure 2.10. This routine examines $\texttt{STATE}_s$ for all scheduled successors, and merges them into one $\texttt{STATE}_I$.

2.6 Correctness of the Algorithm

This section demonstrates that the algorithm presented in the previous section will guarantee that the synchronization state of a register is always cleared before an operation using that register as its *destination* is issued. This is shown by providing more formal definitions for the concepts discussed in previous sections, describing invariants for the algorithm, and then providing a demonstration of correctness.

2.6.1 Basic Definitions

This following provides more formal definitions for many of the concepts introduced in the previous several sections.

B is the complete set of basic blocks in a program.

R is the complete set of available machine registers.

SCHEDULED(b) = TRUE if code has been generated for block b.

FIRST(b, r), $b \in \mathbf{B}, r \in \mathbf{R}$, is the first operation in a block b that references r as *source* and/or *destination*. FIRST(b, r) is one of `READ, WRITE, LOAD`.

LAST(b, r), $b \in \mathbf{B}, r \in \mathbf{R}$, is the last operation in a block b that references r as *source* and/or *destination*. LAST(b, r) is one of `READ, WRITE, LOAD`.

SOURCE(OP(b, r)) = TRUE if r is a *source* for OP(r) in b.

```
for each operation O {
    switch(instruction type of O) {
        case READ:
            if STATE_b(source(O)) = NULL
                STATE_b(source(O)) = GROUNDED
            if STATE_b(target(O)) = NULL
                STATE_b(target(O)) = FULL
            if source(O) ∈ PENDING_b
                remove source(O) from PENDING_b
            break;

        case WRITE:
            if target(O) ∈ PENDING_b {
                issue READ of target(O)
                remove target(O) from PENDING_b
            }
            if STATE_b(target(O)) = NULL
                STATE_b(target(O)) = FULL
            break;

        case LOAD:
            if target(O) ∈ PENDING_b {
                issue READ of target(O)
                if STATE_b(target(O)) = NULL
                    STATE_b(target(O)) = GROUNDED
            }
            else {
                add target(O) to PENDING_b
                if STATE_b(target(O)) = NULL
                    STATE_b(target(O)) = PENDING
            }
            break;

        issue O
}
```

Figure 2.9: **Scheduling Algorithm for a Basic Block.**

```
for all registers r
    STATE_I(r) = GROUNDED;
for all scheduled successors s of b
    for all registers r
        if (STATE_I(r) = PENDING ∨ STATE_s(r) = PENDING)
            STATE_I(r) = PENDING;
        else if (STATE_I(r) = FULL ∨ STATE_s(r) == FULL)
            STATE_I(r) = FULL;
return STATE_I;
```

Figure 2.10: **Algorithm for Merging STATE from Scheduled Successors.**

OP $\in$ {FIRST,LAST}, $r \in \mathbf{R}, b \in \mathbf{B}$.

DESTINATION(OP(b, r)) = TRUE if r is a *destination* for OP(r) in b.

OP $\in$ {FIRST,LAST}, $r \in \mathbf{R}, b \in \mathbf{B}$.

SUCCESSOR(b, s) = TRUE if $b, s \in \mathbf{B} \wedge s$ is a successor of b.

PREDECESSOR(b, p) = TRUE if $b, p \in \mathbf{B} \wedge p$ is a predecessor of b.

Processing of each instruction involves two distinct phases. First, all of the registers which are used as *sources* for the instruction are processed as having been READ. Next, the *destination* register is processed as being either WRITTEN or LOADED depending on the instruction. If the *destination* is PENDING after the first phase of processing, a synchronizing READ must be inserted before this instruction is scheduled.

2.6.2 Definitions for Transitional Information

These definitions can be used to define rules to determine the contents of the sets $\mathtt{STATE}_b$ and $\mathtt{PENDING}_b$. Recall that $\mathtt{STATE}_b$ is the register synchronization information upwardly exposed from a scheduled successor of the current block. If the first reference

to a register performs a READ on the register, then $\texttt{STATE}_b(r)$ for that register is GROUNDED:

$$\text{SOURCE}(\text{FIRST}(b,r)) = \text{TRUE} \rightarrow \texttt{STATE}_b(r) = \texttt{GROUNDED}$$

If the first reference to a register is as *destination*, the operation could be either a WRITE or a LOAD. If it is a WRITE, then $\texttt{STATE}_b(r)$ is FULL, if it is a LOAD, then $\texttt{STATE}_b(r)$ is PENDING:

$$\text{SOURCE}(\text{FIRST}(b,r)) = \text{FALSE} \wedge \text{FIRST}(b,r) = \texttt{WRITE} \rightarrow \texttt{STATE}_b(r) = \texttt{FULL}$$

$$\text{SOURCE}(\text{FIRST}(b,r)) = \text{FALSE} \wedge \text{FIRST}(b,r) = \texttt{LOAD} \rightarrow \texttt{STATE}_b(r) = \texttt{PENDING}$$

It is possible to combine the $\texttt{STATE}_s$ for all scheduled successors of a block once some rules of precedence are defined. If $\texttt{STATE}_s(r) = \texttt{PENDING}$ for any successor, $\texttt{STATE}_I(r) = \texttt{PENDING}$. If $\texttt{STATE}_s(r) \neq \texttt{PENDING}$ for all successors, but $\texttt{STATE}_s(r) = \texttt{FULL}$ for any successor, $\texttt{STATE}_I(r) = \texttt{FULL}$. Otherwise, $\texttt{STATE}_s(r) = \texttt{GROUNDED}$ for all successors, and $\texttt{STATE}_I(r) = \texttt{GROUNDED}$. Using this definition of $\cap$, it is possible to define:

$$\begin{aligned}\texttt{STATE}_I \;=\; & \{\cap_i \texttt{STATE}_{s_i} \mid s_i \in \mathbf{B} \wedge \text{SCHEDULED}(s_i) = \text{TRUE} \\ & \wedge \text{SUCCESSOR}(b, s_i) = \text{TRUE}\end{aligned}$$

There are two rules needed for determining the contents of $\texttt{PENDING}_b$. If the last reference to a register uses the register as *destination*, and the operation is a LOAD, then $\texttt{PENDING}_b(r)$ for that register is TRUE. In all other cases, $\texttt{PENDING}_b(r)$ for that register is FALSE:

$$\begin{aligned}\text{DESTINATION}(\text{LAST}(b,r)) \;=\; & \text{TRUE} \wedge \text{LAST}(b,r) = \texttt{LOAD} \\ & \rightarrow \texttt{PENDING}_b(r) = \text{TRUE} \\ \text{DESTINATION}(\text{LAST}(b,r)) \;=\; & \text{FALSE} \vee \text{LAST}(b,r) \neq \texttt{LOAD} \\ & \rightarrow \texttt{PENDING}_b(r) = \text{FALSE}\end{aligned}$$

For a given block, the initial contents of $\texttt{PENDING}_b$ is the union of the sets $\texttt{PENDING}_p$ from all of the scheduled predecessors of the block b. For this union operation, a register is considered to be in $\texttt{PENDING}_b$, **iff** $\texttt{PENDING}_p(r)$ = TRUE for any scheduled predecessor p. This leads to the definition:

$$\begin{aligned}\texttt{PENDING}_b \;=\; & \{\cup_i \texttt{PENDING}_{p_i} \mid p_i \in \mathbf{B} \wedge \text{SCHEDULED}(p_i) = \text{TRUE} \\ & \wedge \text{PREDECESSOR}(b, p_i) = \text{TRUE}\}\end{aligned}$$

2.6.3 Invariants, Preconditions and Postcondition

The preconditions for the algorithm are:

- **B** contains all of the basic blocks for a procedure.
- The graph containing **B** is properly organized to represent the relationships between all blocks and their predecessors/successors
- Each $b \in \mathbf{B}$ contains a set of operations $\{O | O \in \{\texttt{READ}, \texttt{WRITE}, \texttt{LOAD}\}$
- All operations are legal to issue
- $b \in \mathbf{B} \rightarrow$ SCHEDULED(b) = FALSE

The invariants for this algorithm are:

- $b \in \mathbf{B} \wedge$ SCHEDULED(b) = TRUE $\rightarrow$ All original operations in b have been issued.
- $b \in \mathbf{B} \wedge$ SCHEDULED(b) = TRUE $\rightarrow$ All required **READS** have been inserted in b.
- $b \in \mathbf{B} \wedge$ SCHEDULED(b) = TRUE $\rightarrow$ $\texttt{PENDING}_b$ is correct.
- $b \in \mathbf{B} \wedge$ SCHEDULED(b) = TRUE $\rightarrow$ $\texttt{STATE}_b$ is correct.

The termination condition for the algorithm is:

- $b \in \mathbf{B} \rightarrow$ SCHEDULED(b) = TRUE

The postconditions for the algorithm are:

- $\forall b \in \mathbf{B}$ $\texttt{PENDING}_b$ is correct.
- $\forall b \in \mathbf{B}$ All original operations in b have been issued.
- $\forall b \in \mathbf{B}$ All required READS have been inserted in b.
- $\forall b \in \mathbf{B}$ $\texttt{STATE}_b$ is correct.

That is, the invariants hold for all basic blocks in the program. Recall that the specification for this algorithm is:

- Obtain register state information from scheduled predecessors. Guaranteed by invariants **3** and **4**.
- Generate a schedule for the current block, inserting synchronizing READS where required. Guaranteed by invariants **1** and **2**.
- Examine scheduled successors, insert any READS required for transition from current block into scheduled blocks. Guaranteed by invariants **2** and **3**.
- Ensure that state information made available to all predecessors and successors is correct. Guaranteed by invariants **3** and **4**.

2.6.4 Demonstration of Correctness

Initially, no blocks have been scheduled, and all invariants are trivially preserved. If $\mathbf{B} = \emptyset$, then there are no basic blocks within the current procedure, and the algorithm trivially terminates. Otherwise, there is at least one block which needs to be scheduled and the outer loop denoted by:

while (not all blocks scheduled)

will be entered. This loop condition can be rewritten as:

while ($\{\exists b | b \in \mathbf{B} \wedge$ SCHEDULED(b) = FALSE $\}$)

Since `SelectBlock()` is guaranteed to return a value $\{b | b \in \mathbf{B} \wedge$ SCHEDULED($\mathbf{B}$) = FALSE$\}$, and such a b exists, after execution of this line it can be asserted that: $b \in \mathbf{B} \wedge$ SCHEDULED($\mathbf{b}$) = FALSE.

The next step in the algorithm is to ensure that the set $\texttt{PENDING}_b$ contains only registers that are in $\texttt{PENDING}_p$ of scheduled predecessors p, and contains all such registers. The initialization:

$\texttt{PENDING}_b = \emptyset$;

allows us to assert: $\texttt{PENDING}_b$ contains no registers not present in $\texttt{PENDING}_p$ of a scheduled predecessor of b. If there are no scheduled predecessors of b, then this initial value is correct. If there are scheduled predecessors, the following code will ensure that $\texttt{PENDING}_b$ contains the proper set of registers:

for all predecessors p of b
    if (predecessor scheduled)
        $\texttt{PENDING}_b = \texttt{PENDING}_b \cup \texttt{PENDING}_p$;

This can be expressed as:

$(\{\forall\, p \mid p \in \mathbf{B} \wedge$ PREDECESSOR$(b, p) =$ TRUE $\})$
    if (SCHEDULED(p) = TRUE)
        $\texttt{PENDING}_b = \texttt{PENDING}_b \cup \texttt{PENDING}_p$;

This loop will terminate as it iterates only once for each predecessor of b, and the set of predecessors is finite. After execution of this loop the following can be asserted:

$$r \in \mathbf{R} \wedge r \in \texttt{PENDING}_b \leftrightarrow r \in \texttt{PENDING}_p \wedge \text{SCHEDULED}(p) = \text{TRUE} \wedge \text{PREDECESSOR}(b, p) = \text{TRUE}$$

At this point, the initial computation of $\texttt{PENDING}_b$ is complete.

It will be demonstrated in Section 2.6.4 that, after the call:

`Schedule(`b`)`;

the following can be asserted: all original operations in b have been issued; all required READS have been inserted in b; $\texttt{PENDING}_b$ is correct up to this point; and $\texttt{STATE}_b$ is correct up to this point.

It will be shown in Section 2.6.4 that, following the call:

$\texttt{STATE}_s$ = JoinSuccessorStates(b);

the following can be asserted: $\texttt{STATE}_s$ is correct.

Next, an iteration over all registers is performed to identify any registers which are in $\texttt{PENDING}_b$, but are not grounded in $\texttt{STATE}_s$. In addition, any registers for which $\texttt{STATE}_b(r) = \texttt{NULL}$ are set to $\texttt{STATE}_s(r)$:

```
for all registers r
    if r ∈ PENDING_b ∧ (STATE_s(r) = PENDING ∨ STATE_s(r) = FULL)
        issue READ of r
        remove r from PENDING_b
        if (STATE_b(r) = NULL)
            STATE_b(r) = GROUNDED;
    if (STATE_b(r) = NULL)
        STATE_b(r) = STATE_s(r);
```

This loop will terminate as each iteration examines one register and the number of registers is finite. After this code has executed, it can be asserted that: $\texttt{STATE}_b$ is correct; all necessary READs to compensate for transitions between this block and scheduled successors have been issued; $\texttt{PENDING}_b$ is correct. Finally, SCHEDULED(b) is updated using:

SCHEDULED(b) = TRUE

And all of the invariants hold for the current block.

As this loop will be executed exactly once for each block $b \in \mathbf{B}$, and there are a finite number of blocks in $\mathbf{B}$, the loop will terminate. After each iteration of the loop SCHEDULED(b) holds for one more block than it had on the previous iteration. Therefore, when the loop terminates the postcondition:

$b \in \mathbf{B} \rightarrow$ SCHEDULED(b) = TRUE

has been satisfied.

Correctness of the routine `Schedule()`

Recall that the required postconditions of this routine are:

- All original operations in **B** have been issued.
- All required `READS` have been inserted in **B**.
- $\texttt{PENDING}_b$ is correct up to this point.
- $\texttt{STATE}_b$ is correct up to this point.

Recall also, that the following preconditions have been shown to hold on entry to this routine:

- $b \in \mathbf{B} \wedge$ SCHEDULED(b) = FALSE
- $\texttt{PENDING}_b$ is correct up to this point.

This entire routine is in the form of a single loop denoted by:

for each operation O

As a block must have a finite number of operations, and each operation is dealt with in a single loop iteration, this loop will terminate. As the bottom of this loop is the instruction:

issue O

it can be trivially asserted that all original operations in b are issued.

Depending on the type of operation currently being handled, one of three clauses is executed. By showing that each clause properly handles one type of operation, it will be demonstrated that all operations will be properly handled.

The only item of concern for processing a `READ` is to ensure that following the `READ`, its *source* register is not in $\texttt{PENDING}_b$. In terms of $\texttt{STATE}_b$, if this is the first reference to the *source* register, its state is set to `GROUNDED`, and if this is the first

reference to the *destination* register its state is set to FULL. This is all accomplished with the clause:

```
if STATE_b(source(O)) = NULL
    STATE_b(source(O)) = GROUNDED
if STATE_b(target(O)) = NULL
    STATE_b(target(O)) = FULL
if source(O) ∈ PENDING_b
    remove source(O) from PENDING_b
```

Following this clause and the issuing of the instruction at the bottom of the loop, it can be asserted that if the first operation is a READ, all postconditions hold at the bottom of the loop after the first iteration.

When dealing with a WRITE, if the *target* register is in $\texttt{PENDING}_b$, it is necessary to insert a READ of this register. Following this insertion, the register is removed from $\texttt{PENDING}_b$. In either case, if this is the first reference to the *target* variable, its state is set to FULL. These manipulations are performed by:

```
if target(O) ∈ PENDING_b {
    issue READ of target(O)
    remove target(O) from PENDING_b
}
if STATE_b(target(O)) = NULL
    STATE_b(target(O)) = FULL
```

Following this clause and the issuing of the instruction at the bottom of the loop, it can be asserted that if the first operation is a WRITE, all postconditions hold at the bottom of the loop after the first iteration.

When dealing with a LOAD, if the *target* of the LOAD is in $\texttt{PENDING}_b$, it is necessary to insert a READ of this register. The register is not removed from $\texttt{PENDING}_b$ however, as the *target* of the LOAD currently being processed must be in $\texttt{PENDING}_b$ once the current instruction has been issued. If we are to issue this read of the *target* register,

and this is the first reference to this register, its state is set to GROUNDED. If the *target* is not in $\texttt{PENDING}_b$, it is added to $\texttt{PENDING}_b$. If this is the first reference to this register, its state is set to PENDING. This logic is encoded as:

```
if target(O) ∈ PENDING_b {
    issue READ of target(O)
    if STATE_b(target(O)) = NULL
        STATE_b(target(O)) = GROUNDED
}
else {
    add target(O) to PENDING_b
    if STATE_b(target(O)) = NULL
        STATE_b(target(O)) = PENDING
}
```

Following this clause, and the issuing of the instruction at the bottom of the loop, it can be asserted that if the first operation is a LOAD, all postconditions hold at the bottom of the loop after the first iteration.

It can now be asserted that if the instruction is any of READ, WRITE, LOAD, following the issuing of the instruction at the bottom of the loop all postconditions hold after the first iteration. This can be generalized to the statement that all postconditions will hold at the bottom of any loop iteration, in particular the final loop iteration. Therefore, all postconditions hold at the end of this routine.

Correctness of the routine `JoinSuccessorStates()`

Recall that the requirement of this routine is that it has as its postcondition:

$\texttt{STATE}_s$ is correct

To be able to discuss this, it is necessary that some description be provided as to what it means for this condition to hold. If $\texttt{STATE}_s(r)$ for a register is PENDING for any scheduled successor s, then it is irrelevant what $\texttt{STATE}_s(r)$ for that register is for any other scheduled successor s. The register must be treated as if it is PENDING for all successors. If $\texttt{STATE}_s(\texttt{r}) \neq \texttt{PENDING}$ for all scheduled successors, but it is FULL for

one or more successors, then it is treated as being FULL for all successors. If neither of these conditions are met, then the register is treated as being GROUNDED for all successors.

The preconditions for this routine are:

- $b \in \mathbf{B}$
- The information about successors to b is valid

The above description implies that the default setting for a register, unless it is overridden by a value from a scheduled successor, is GROUNDED. Therefore, the first operation in the routine is:

for all registers r
    $\mathtt{STATE}_s(r) = \mathtt{GROUNDED}$;

This loop will terminate as it iterates once per register and there are a finite number of registers. If there are no scheduled successors of b, then the routine terminates here, and the returned $\mathtt{STATE}_s$ is GROUNDED for all registers. This is the correct return value for this circumstance. If there are scheduled successors, the loop denoted by:

for all scheduled successors of b

will be entered. This can be rewritten as:

$(\forall\ s \mid s \in \mathbf{B} \wedge \text{SCHEDULED}(s) = \text{TRUE} \wedge \text{SUCCESSOR}(b, s) = \text{TRUE})$

This loop will terminate as there must be a finite number of successors to a block. Within this loop, there is another loop which is denoted by:

for all registers r

This loop will also terminate as the number of registers is finite. The body of this loop is:

```
if (STATE_s(r) = PENDING ∨ STATE_s(r) = PENDING)
    STATE_s(r) = PENDING;
else if (STATE_s(r) = FULL ∨ STATE_s(r) = FULL)
    STATE_s(r) = FULL;
```

The first check shown here ensures that if any scheduled successor has $\texttt{STATE}_s(r) = \texttt{PENDING}$, then the set returned from this routine will have $\texttt{STATE}_s(r) = \texttt{PENDING}$. Similarly, if none of the scheduled successors has $\texttt{STATE}_s(r) = \texttt{PENDING}$, but one or more of them have $\texttt{STATE}_s(r) = \texttt{FULL}$, then the set returned from this routine will have $\texttt{STATE}_s(r) = \texttt{FULL}$. If none of the scheduled successors have either $\texttt{STATE}_s(r) = \texttt{PENDING}$ or $\texttt{STATE}_s(r) = \texttt{FULL}$, then the set returned from this routine will have the default value of $\texttt{STATE}_s(r) = \texttt{GROUNDED}$. Upon assignment of the return value from this routine to $\texttt{STATE}_s$, it can be asserted that $\texttt{STATE}_s$ is correct.

2.6.5 Managing Transfer of Control

The preceding algorithm and proof do not address the issue of managing register state when a transfer of control, such as a function call or interrupt, occurs. It is assumed that interrupts are not an issue as they will either only use special hardware registers set aside for their use, or will first read any registers that they are going to write to preserve their initial values. The interrupt handler must take action to ensure that no registers which were not PENDING on entry to the handler are PENDING when control is returned to the user program.

The algorithm relies on all registers being in a known state upon entry to a function. In order to support separate compilation of source files and the use of libraries, it is necessary to ensure that no registers are PENDING prior to issuing a function call. In a system where all source was required to be in a single input file, it would be possible to extend the interblock exchange of information to also handle interprocedural exchange. It is also possible to design a system that uses this interprocedural exchange of information by importing information from previously

compiled images, but that is outside the scope of this thesis.

Chapter 3 Algorithm Modifications for Trace-Scheduling

The previous chapter dealt with a traditional compiler which works with basic block scheduling. The Multiflow compiler being used for the M-Machine work uses a trace scheduling algorithm [17, 37]. In terms of the SRS algorithm, the major difference between basic blocks and traces is that, unlike a basic block, a trace can have multiple entry and exit points. This chapter explains the changes that are necessary to the algorithm presented in Section 2.5 to accommodate trace scheduling; and discusses the impact of these changes on the demonstration of correctness.

3.1 Algorithm Modifications for Trace Scheduling

The major changes to the SRS algorithm for accomodating trace-scheduling deal with transitions between traces. First, changes need to be made to the management of the data transferred between traces. For the basic block algorithm it made sense to associate this information with each block. When dealing with trace-scheduling a more natural association is to be found with the edges between traces. This provides a natural way of handling the multiple entry and exit points which can exist within a trace. One of the properties of these edges is that they only have one entry point and one exit point.

3.1.1 Processing a Trace

For each cycle in the schedule being generated, there are three phases to the processing. Initially, it is necessary to manage any edges that enter the trace at the current cycle. Each instruction within the current cycle must be processed. Finally, the management of edges leaving the trace after the current cycle must be managed.

At the beginning of each cycle, a check is made to determine if any edges join the current trace at this cycle. If edges do join at this cycle, it is necessary to either incorporate PENDING information from the edge if it has been scheduled, or to attach STATE information to the edge if it hasn't been scheduled. The final register states from any scheduled predecessors are combined.

If a register is PENDING in any scheduled predecessor, it must be treated as if it is PENDING in all scheduled predecessors. As GROUNDED applies only to upwardly-exposed state it is not relevant here. For any given instruction, it is necessary to have incorporated information about all preceding instructions which have already been scheduled.

When STATE is attached to an edge this does not mean that the current values of $\texttt{STATE}_t$ that have already been computed for this trace are copied into the edge. It means that a new $\texttt{STATE}_e$ is initialized, and all further computation of STATE that occurs in the process of scheduling the current trace must update not only $\texttt{STATE}_t$ for the trace, but all $\texttt{STATE}_e$ for unscheduled edges that have already joined the trace. This process will ensure that when the edge is scheduled it is presented with a clear picture of what will happen to registers following the transition from the edge into the adjoining trace.

For each instruction within the current cycle, several stages of processing are performed. All instruction source operands that are in registers are identified, and those registers are marked FULL. The state of the destination register, if any, is checked. If the destination register is PENDING, a READ on the destination register is inserted

prior to the current instruction. If the instruction is a WRITE the destination register is marked FULL. If the instruction is a LOAD the destination register is marked PENDING.

If there are unscheduled edges leaving the trace, $\texttt{PENDING}_t$ is attached to the edge to be used when it is scheduled. If there are scheduled edges leaving the trace, a check is made to determine if any synchronizing operations need to be inserted before the next instruction is emitted.

At the end of each cycle, management of edges that split from the current trace are handled. If these edges have already been scheduled, a check is made to determine if any registers need to be READ prior to transitioning to the edge. If the adjoining edge has not been scheduled, then a copy of $\texttt{PENDING}_t$ is placed on the edge. If there are multiple edges, all of the unscheduled edges should be handled first. This maximizes the distance between initiating a LOAD operation and the first off-trace access to the targeted register.

3.1.2 Trace Algorithm

One of the interesting aspects of managing traces is that there is the possibility of edges joining to and splitting from a trace at every cycle of the schedule. This increases the complexity of the routine for scheduling the trace, but vastly simplifies the top level of the algorithm. The top-level code necessary is shone in Figure 3.1, all that remains from Figure 2.8 is some trivial initialization, and marking the completed trace as scheduled. The routine SelectTrace has replaced SelectBlock, but has an equivalent definition: it returns a trace that has not yet been scheduled. $\texttt{PENDING}_t$ is initialized to empty, and $\texttt{STATE}_t$ is set to NULL for all registers. The trace is scheduled, and then marked as scheduled.

Figure 3.2 shows the algorithm for scheduling a trace. This combines the mangement of transitions between traces shown in Figure 2.5, and the state transitions shown in Figure 2.6. This algorithm also manages the sets $\texttt{PENDING}_t$ and $\texttt{STATE}_t$ for

```
while (not all traces scheduled)
    t = SelectTrace();
    PENDING_t = ∅;
    for all registers r
        STATE_t(r) = NULL

    Schedule(t);

    Mark t as scheduled
```

Figure 3.1: **General Outline of Trace Algorithm.**

the current trace, as well as $\texttt{STATE}_e$ for all unscheduled edges that join the current trace. The basic structure of the algorithm is to iterate over all cycles in the trace, executing a prologue, loop body, and epilogue.

The loop prologue manages incoming edges. For all scheduled incoming edges, the information from $\texttt{PENDING}_e$ is incorporated into $\texttt{PENDING}_t$. For unscheduled incoming edges, $\texttt{STATE}_e$ is created and associated with the edge.

The loop body processes all operations within the current cycle ensuring that they are all scheduled and all necessary READS are inserted. the loop body also updates $\texttt{PENDING}_t$, $\texttt{STATE}_t$ and $\texttt{STATE}_e$ based on the operations being scheduled. Figure 3.2 shows the manipulation of $\texttt{PENDING}_t$ in accordance to the rules illustrated in Figure 2.6. This same set of rules controls the management of $\texttt{STATE}_t$ and $\texttt{STATE}_e$ shown in Figure 3.3, which is called once for $\texttt{STATE}_t$ and once for each $\texttt{STATE}_e$ from unscheduled edges that have previously joined the current trace.

The epilogue manages outgoing edges. First all of the unscheduled edges have $\texttt{PENDING}_t$ copied into them. This allows any necessary register synchronization to be delayed as long as possible on these edges. For the scheduled outgoing edges, $\texttt{STATE}_e$ needs to be reconciled with $\texttt{PENDING}_t$. Each register is checked. Any register which is in $\texttt{PENDING}_t$ but is not GROUNDED in $\texttt{STATE}_e$ needs to be READ before the transition

```
for each cycle C {
    for each edge e joining t at C {                    Begin Prologue: Process incoming edges
        if (edge e scheduled)                           For scheduled edges, add PENDING_e into PENDING_t
            PENDING_t = PENDING_t ∪ PENDING_e
        else
            Create STATE_e                              For unscheduled edges, create STATE_e
    }                                                   End Prologue: All incoming edges have been processed

    for each operation O {                              Begin loop body
            updateState(O, t, t);                       Update STATE_t
        for each edge e that joins trace above O
            updateState(O, t, e);                       Update STATE_e

        switch(instruction type of O) {
            case READ:                                  After a READ, the source is GROUNDED
                if source(O) ∈ PENDING_t
                    remove source(O) from PENDING_t
            break;

            case WRITE:                                 A WRITE can't proceed if target is PENDING
                if target(O) ∈ PENDING_t {
                    issue READ of target(O)             GROUND the target
                    remove target(O) from PENDING_t
                }
            break;

            case LOAD:                                  A LOAD can't proceed if target is PENDING
                if target(O) ∈ PENDING_t
                    issue READ of target(O)             GROUND the target
                else
                    add target(O) to PENDING_t          Target must be in PENDING_t once the LOAD has issued
            break;
        }
        issue O
    }                                                   End Loop Body

    for each unscheduled edge e splitting from t at C   Begin Epilogue: Process outgoing edges
        Attach PENDING_t to e                           For unscheduled edges, attach PENDING_t
    for each scheduled edge e splitting from t at C
        for all registers r                             For scheduled edges reconcile PENDING_t with STATE_e
            if (r ∈ PENDING_t AND (STATE_e(r) ≠ GROUNDED)
                issue read of r
                remove r from PENDING_t                 End Epilogue: All outgoing edges have been processed
}
```

Figure 3.2: **Scheduling Algorithm for a Trace.** This algorithm is divided into three sections. The prologue manages all incoming edges, either incorporating information from $\texttt{PENDING}_e$ into $\texttt{PENDING}_t$, or creating $\texttt{STATE}_e$, depending on whether or not the edge is scheduled. The loop body maintains $\texttt{STATE}_t$ and $\texttt{STATE}_e$ for all edges that have previously joined the trace. In addition, $\texttt{PENDING}_t$ is maintained while all operations are issued and all required READs are inserted. Finally, the epilogue manages transition to edges leaving the trace at the end of the current cycle. Either $\texttt{PENDING}_t$ is copied into the edge, or READs are inserted to reconcile $\texttt{PENDING}_t$ with $\texttt{STATE}_e$, again depending on whether or not the edge has been scheduled.

```
updateState(Operation, t, e) {
    switch(instruction type of O) {
        case READ:
            if STATE_e(source(O)) = NULL
                STATE_e(source(O)) = GROUNDED
            if STATE_e(target(O)) = NULL
                STATE_e(target(O)) = FULL
        break;

        case WRITE
            if STATE_e(target(O)) = NULL
                STATE_e(target(O)) = FULL
        break;

        case LOAD
            if target(O) ∈ PENDING_t {
                if STATE_e(target(O)) = NULL
                    STATE_e(target(O)) = GROUNDED
            }
            else if STATE_e(target(O)) = NULL
                STATE_e(target(O)) = PENDING
        break;
    }
}
```

Figure 3.3: **Routine to Manage State During Scheduling.** This algorithm tracks the first reference to each register relative to when the edge e joined the current trace t. This processing simply encodes the state transitions shown in Figure 2.6, with the additional checks to ensure that this is the first reference to the relevant register.

to the edge can occur. Ideally these READs are made conditional so that they are only issued if the outgoing edge is going to be executed. This is only practical in a machine with predicated instructions.

3.1.3 Correctness Modifications for Trace Scheduling

Most of the changes to the algorithm to accommodate trace-scheduling have little impact on the previously demonstrated correctness of the algorithm. The one area which bears some examination is the code that occurs at the beginning and end of each cycle of scheduling. These pieces of code mirror in many respects the code at the beginning and end of each basic block/trace.

At the beginning of each cycle, a check is made to determine if any edges join the current trace at this cycle. If edges do join at this cycle, it is necessary to either incorporate PENDING information from the edge if it has been scheduled, or to attach STATE information to the edge if it hasn't been scheduled. In the first case, this is directly analogous to incorporating $\texttt{PENDING}_p$ from all predecessors to a given trace into $\texttt{HOT}_t$ prior to beginning scheduling of the trace. For any given instruction, it is necessary to have incorporated information about all preceding instructions which have already been scheduled. For the latter case, some clarification is needed. When STATE is attached to an edge this does not mean that the current values of $\texttt{STATE}_t$ that have already been computed for this trace are copied into the edge. It means that a new $\texttt{STATE}_e$ is initialized, and all further computation of STATE that occurs in the process of scheduling the current trace must update not only $\texttt{STATE}_t$ for the trace, but all $\texttt{STATE}_e$ for unscheduled edges that have already joined the trace. This process will ensure that when the edge is scheduled it is presented with a clear picture of what will happen to registers following the transition from the edge into the adjoining trace.

At the end of each cycle, management of edges that split from the current trace are handled. If these edges have already been scheduled, a check is made to determine

if any registers need to be READ prior to transitioning to the edge. If the adjoining edge has not been scheduled, then a copy of $\texttt{PENDING}_t$ is placed on the edge. Relative to this edge, this is equivalent to placing $\texttt{PENDING}_t$ into an adjoining trace. If there are multiple edges all of the unscheduled edges should be handled first, to maximize the distance in between initiating a LOAD operation and the first off-trace access to the targeted register.

3.2 Implementation of Software Register Synchronization

As part of the M-Machine software effort, the Scalable Concurrent Programming Laboratory at Caltech has acquired the source code to the Multiflow compiler and is using it as a platform for compiler development and experimentation. The first phase of the project – porting it the MAP instruction set, improving code quality, incorporating optimizations, and using it as a vehicle for architectural evaluation – has been completed as part of this thesis research. The compiler generates code for a single MAP cluster and although fully optimized, most of the standard optimizations, including common subexpression elimination, copy propagation, and loop invariant code motion, are operational.

The second phase of the compiler modifications entails developing and implementing algorithms for effective use of the loosely-coupled clusters of functional units within the MAP chip. This includes inserting appropriate synchronization and communication instructions into the streams that execute on different clusters but are intended to work together, sharing data among their register files. The MAP's non-blocking memory system with unpredictable latencies and its capability for remote register writes has caused the issue of WAW hazards to arise in both phases of compiler development. The work described in this section focuses on Software Register

Synchronization (SRS), which eliminates the WAW hazards found while compiling code for a single cluster.

3.2.1 Compiler Scoreboarding

In the MAP, a hardware register scoreboard is provided to eliminate RAW hazards. In order to avoid additional hardware expense, the M-Machine compiler is responsible for detecting WAW hazards and converting them into RAW hazards so that the hardware register scoreboard will prevent them. Thus if the compiler finds an instruction that writes to a register that is already pending, it inserts an instruction that reads from that register so that the thread will stall until the WAW hazard is resolved.

There are two methods of performing this synchronization, *register barriers* and *memory barriers*. The first is on a fine grained register basis in which an instruction to read from a single register, a *register barrier*, is inserted. This is useful when a WAW hazard on an individual register is discovered. The second method is useful for eliminating WAW hazards across control flow changes such as procedure calls and returns. The MAP provides a memory barrier (`mbar`) instruction which stalls until all outstanding memory references to the cluster have returned. This allows synchronization with a single instruction on groups of WAW hazards that are due to references to the non-blocking memory system.

In order to detect WAW hazards, the compiler maintains a software scoreboard that monitors reads and writes to registers during compilation. When an instruction is scheduled, its destination register is marked pending. When a subsequent instruction that reads from the register is scheduled, the register is marked full; the hardware is responsible for ensuring RAW synchronization at runtime. Due to the uncertainty of both memory latencies in a non-blocking memory system, and communication timing between loosely-coupled processors, the compiler assumes that no previous register writes have completed unless there is an intervening read from that register. This

provides a clear goal for the software scoreboard: monitor all register accesses and insert intervening register reads when a WAW hazard is detected.

Because WAW hazards are largely due to physical resource constraints (not enough registers), the compiler's register scoreboard and monitoring is performed during the register allocation phase of compilation. This has the additional benefit that the register allocator can be biased away from using a register that would create a WAW hazard. If the compiler increases the distance between conflicting writes, then the register read that eliminates the WAW hazard may not need to stall at runtime, resulting in only a single cycle overhead. The initial implementation of SRS, evaluated in this thesis, performs the full algorithm functionality within traces, but does not copy register state information across trace boundaries. When transitions off of the current trace are detected, synchronizing operations may be inserted to force PENDING registers into the FULL state prior to entry to the adjoining trace.

One of the roles of hardware RAW management is handling antidependencies. As the base scheduling paradigm for the Multiflow compiler is a VLIW machine containing no hardware dependence checking, the scheduler will generate code which properly manages antidependencies. The definition of an antidependence [30] is that the first instruction involved is a `READ`. The synchronizing nature of this `READ` assures that the first operation will have completed before the second one is issued.

Figure 3.4 shows an example of a WAW hazard in a small code sequence that is resolved by the compiler scoreboard algorithm. The `move` instruction is the intervening read; `r0` is used as a target of the synchronization instruction as `r0` is mapped to zero and writes to it do not actually occur.

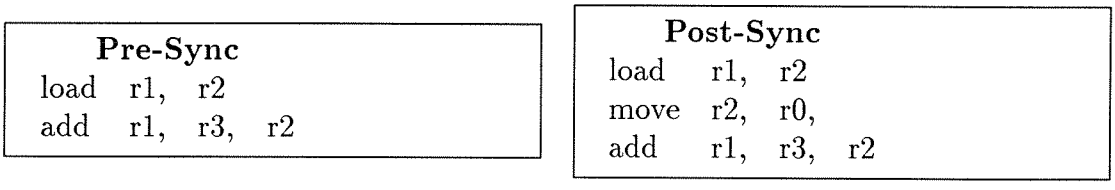


Figure 3.4: **Code Transformation to Convert a WAW hazard into a RAW Hazard.** The pre-synchronized sequence clearly has a WAW hazard as the load may complete after the add. In the post-synchronized sequence, a synchronizing instruction has been added. The move operation blocks until the load completes, after which the add is able to execute.

Chapter 4 Algorithm Modifications for a Partitioned Register File

The previous two chapters discussed managing register synchronization for a monolithic register file. As discussed in Chapter 1, future systems, including the MIT M-Machine, will have partitioned register files. There are two critical differences between these future architectures and VLIW [20] machines. These new architectures utilize dynamic data dependence checking, as opposed to the static scheduling employed by a VLIW; and the functional units will not necessarily execute in lockstep. This chapter discusses the changes needed to the register synchronization algorithm to manage such an architecture.

Several variations on this architecture are considered. The first variant, referred to as `VLIW+`, is a system with processors operating in lockstep, and a known transfer latency for remote register writes. In the `VLIW+` system, when the processor stalls awaiting completion of a memory operation, the register transfer pipeline also stalls. The second variant to be considered is referred to as `VLIW-Unknown`, and differs from `VLIW+` in two respects. First, the transfer latency for remote registers writes is variable, and is therefore unknown at compile-time; and the transfer pipeline continues to operate even if the machine stalls awaiting completion of a memory operation. Finally, the `MAP` is considered. In this system the processors do not run in lockstep relative to each other; the transfer latency is unknown; and the only portion of the machine that stalls awaiting completion of a `LOAD` operation is the cluster of functional units awaiting the result of the `LOAD`.

In a true VLIW architecture most of the latencies for operations are known at

compile time. This allows the compiler to statically generate a schedule that will execute correctly. One caveat on such a schedule is that it contains a deadline for the completion of memory operations. If the memory subsystem is unable to meet this deadline the entire machine stalls until the memory operation completes. On a machine such as the M-Machine a stall could last in excess of one hundred instructions if the data needs to be fetched from a remote processor. A second caveat on the VLIW static schedule is that it has to assume a deadline for the completion of writes to remote register files. Again, the entire machine would have to stall if the deadline was not met.

While both of these issues are relevant on architectures such as the M-Machine, they do not require full-machine stalls if a deadline is not met. The dynamic dependence checking allows stalling to be delayed until the data is actually needed; and stalls, when they occur, do not need to stop the entire machine, only the portion actually awaiting the delivery of the data. On the M-Machine this would typically be $\frac{1}{3}$ of the processor for a given stall.

The management of memory operations detailed for the monolithic register file in Chapter 3 continues to be appropriate for the partitioned register file. There is an additional case that needs to be managed for partitioned register files: the issuing of an instruction whose target register is in a remote register file. Section 4.1 addresses managing this case for the `VLIW+` architecture. Section 4.2 discusses the `VLIWUnknown` architecture. Section 4.3 discusses managing this issue for the `MAP` chip.

4.1 VLIW+

For a true VLIW architecture, the register synchronization provided by this algorithm is part of the normal VLIW instruction scheduling. This section addresses working with a machine where memory latencies could be high enough that stalling

the machine when a memory completion deadline was missed could severely decrease processor performance. In order for this scheduling to work, the transfer latency has to be maintained as a fixed number of cycles. When the processors stall awaiting completion of a memory operation, the register transfer pipeline also stalls. The situation in which the transfer latency is unknown is covered in Section 4.2.

The scenario being covered in this section is shown in Figure 4.1. In this example, the remote register `WRITE` has a latency of one cycle (the result is available in the remote register at the beginning of the second cycle after the `WRITE` is initiated). Cluster 0 issues a `LOAD` whose target is register i7 **(1)**. Sometime thereafter, Cluster 1 issues an add that will `WRITE` i7 on cluster 0 (denoted cl0.i7) **(2)**. In the cycle between the initiation and completion of this `WRITE`, a synchronizing `READ` is performed by Cluster 0 on register i7 **(3)**. This guarantees that the next access to cl0.i7 **(4)** reads the value generated by **(2)**.

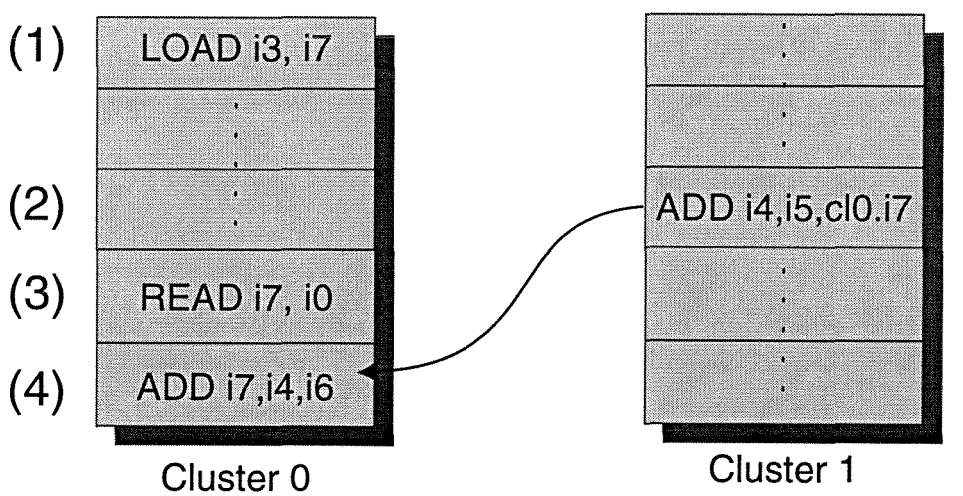


Figure 4.1: **Remote Register Write for VLIW+ with Transfer Latency of One Cycle.**

If the transfer latency is zero (the result is available in the remote register at the beginning of the first cycle after the `WRITE` is initiated) this case is essentially the same as a register `WRITE` in a monolithic register file. This is shown in Figure

4.2. Prior to issuing the WRITE **(2)** a check is made to determine the state of the target register. If it is PENDING, a synchronizing READ **(3)** must be issued prior to the WRITE. As described in Chapter 1, it is not trivial to manage the synchronization of these WRITEs in hardware. One of the advantages of implementing the scoreboard in software is that it is not bound by any of the VLSI constraints that make it prohibitive to implement in hardware.

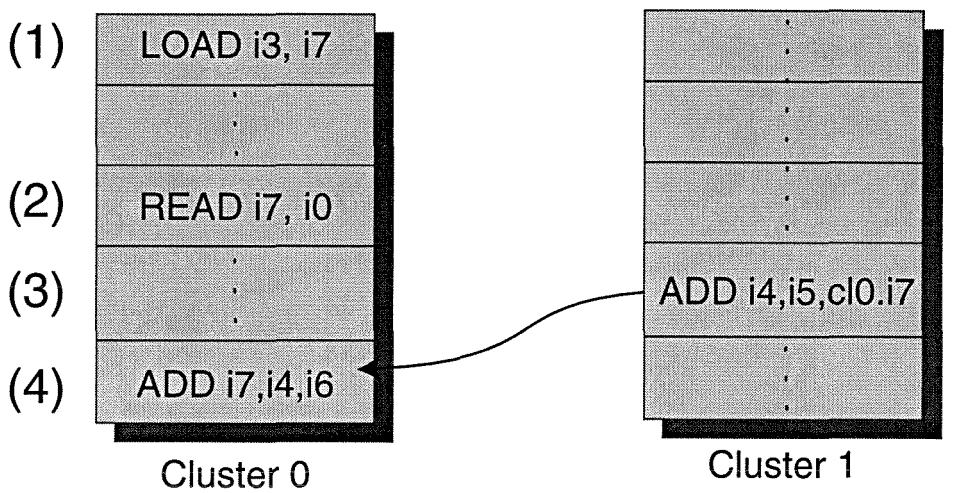


Figure 4.2: **Remote Register Write for VLIW+ with Zero-Latency Transfer.**

Managing LOAD operations that target a remote register is the same as managing non-zero latency transfers in an architecture which does not prevent completion of a remote transfer while the processors are stalled. If two operations are allowed to be in-flight at the same time, there is a race condition; it is not possible to predict which operation will complete first. In order to prevent this situation, it is necessary to guarantee that any previous operation targeting a given register has completed before issuing a new one. This is done by treating a LOAD with a remote target identically to a LOAD with a local target. If the state of the target register is PENDING, a synchronizing READ must be issued prior to issuing the LOAD. If the target register is not PENDING, it is marked PENDING once the LOAD has been issued.

4.1.1 Algorithm Changes for VLIW+

In addition to performing the functionality described in Chapters 2 and 3, the algorithm also needs to maintain information about outstanding remote transfers. This information will be represented as the set `INCOMING`. Each member I of `INCOMING` consists of two elements: `target(I)` which is the register to be written, and `deadline(I)` which is the cycle number in which the operation will complete. As it is possible for there to be pending incoming transfers from an adjacent previously scheduled trace, `INCOMING` is treated similarly to `PENDING`. The union operation for `INCOMING` is to add all incoming transfers into the existing list. Duplicate writes to the same register in a given cycle are discarded. The new general outline of the algorithm is shown in Figure 4.3. The basic structure is essentially the same; JoinSuccessorStates and SelectTrace are unchanged. The algorithm for scheduling a single trace is different, and there is the addition of `INCOMING`. On entry to the trace the handling of `PENDING` and `INCOMING` are identical, although each set has its own union operation.

On exit from a trace, special management is required to manage pending transfers that will complete within an already scheduled successor, and will conflict with a `PENDING` register. If this situation is detected, the compiler must take some action to ensure that there is no conflict between a remote transfer and a memory operation. This processing is encapsulated in the routine StitchIncoming. As the functioning of the scheduler described in Section 1.6.3 prevents this situation from arising, details of StitchIncoming are not provided. In particular, the scheduler would move the *join* point below the problematic register access, and move the intervening code, and management of the remote transfer, onto the newly created edge.

The modified version of the routine Schedule is shown in Figure 4.4. In addition to the functionality shown in Figure 3.2, it maintains `INCOMING` when edges join to or split from the current trace. At the beginning of each cycle `INCOMING` is checked for transfers that will complete in the following cycle. If any such transfers exist, and

```
while (not all traces scheduled)
    t = SelectTrace();
    PENDING_t = ∅;
    INCOMING_t = ∅;
    for all predecessors p of t
        if (predecessor p scheduled)
            PENDING_t = PENDING_t ∪ PENDING_p;
            INCOMING_t = INCOMING_t ∪ INCOMING_p;

    Schedule(t);
    STATE_s = JoinSuccessorStates(t);

    for all registers r
        if (r ∈ PENDING_t AND (STATE_s(r) ≠ GROUNDED)
            issue read of r
            remove r from PENDING_t
            if (STATE_t(r) = NULL)
                STATE_t(r) = GROUNDED;
        if (STATE_t(r) = NULL)
            STATE_t(r) = STATE_s(r);

    if (INCOMING_t ≠ ∅)
        StitchIncoming(t)
    Mark t as scheduled
```

Figure 4.3: **General Outline of Algorithm for VLIW+.**

the target register is PENDING, a synchronizing READ of the target register is issued in the current cycle. As with Figure 4.3, there is a problem if there are pending transfers that will complete within an already scheduled successor, and will conflict with a PENDING register. If this situation is detected, extra cycles must be added into the current schedule at the split point to ensure that there is no conflict between the transfer and a memory operation. Once again, this processing is encapsulated in the call to StitchIncoming, and managed by the scheduler.

4.1.2 Correctness Modifications for VLIW+

This section describes the changes which need to be made to the proof of correctness presented in Chapter 3 to demonstrate the correctness of the algorithm presented in Section 4.1. All of the changes to the demonstration of correctness involve the proper management of INCOMING.

Recall that the primary requirement for PENDING is that it is correct at all points in the computation. The correctness of this set is defined as containing only registers that are in the state PENDING, and containing all such registers. A similar requirement can be defined for INCOMING. This set is to be correct at all points in the computation. This correctness is defined as the set containing information about all outstanding remote transfers at each point in the computation.

Initially $\texttt{INCOMING}_t$ is initialized to $\emptyset$. If there are no scheduled predecessors to this trace this is the correct value. If there are scheduled predecessors, the list of outstanding remote transfers from each of this predecessors is incorporated into $\texttt{INCOMING}_t$. This is done using the same loop that incorporates information from scheduled predecessors into $\texttt{PENDING}_t$. The only change to this loop is the addition of the line:

$$\texttt{INCOMING}_t = \texttt{INCOMING}_t \cup \texttt{INCOMING}_p$$

The $\cup$ operator here adds the entire contents of $\texttt{INCOMING}_p$ to $\texttt{INCOMING}_t$. The

```
for each cycle C
    for each edge e joining t at C
        if (edge e scheduled) {
            PENDING_t = PENDING_t ∪ PENDING_e
            INCOMING_t = INCOMING_t ∪ INCOMING_e
        }
        else {
            Attach STATE_t to e
        }
(1)     for each member I of INCOMING_t
        if deadline(I) == C + 1 {
(2)         if target(I) ∈ PENDING_t
                issue READ of target(I)
                remove target(I) from PENDING_t
        }
(3)     else if deadline(I) == C
            remove I from INCOMING_t

    for each operation O {
        :
        issue O
    }
    for each edge e splitting from t at C
        if (edge e scheduled) {
            for all registers r
                if (r ∈ PENDING_t AND (STATE_e(r) ≠ GROUNDED)
                    issue read of r
                    remove r from PENDING_t
            if (INCOMING_t ≠ ∅)
                StitchIncoming(t)
        }
        else {
            Attach PENDING_t to e
            Attach INCOMING_t to e
        }
```

Figure 4.4: **Scheduling Algorithm for VLIW+.**

only special case is two outstanding transfers with the same target and the same deadline. If this case occurs, only one of the records is maintained. As the records are identical, and will require the same processing to manage, this does not affect the integrity of the set. This operation will ensure that $\texttt{INCOMING}_t$ is correct after it has been issued.

As long as the routine StitchIncoming preserves the required conditions, then the processing at the end of Figure 4.3 will maintain these conditions.

4.1.3 Management of INCOMING while Scheduling a Trace

The management of INCOMING at the beginning and end of each cycle is the same as the handling for entry to and exit from the trace itself.

At the beginning of each cycle it is necessary to examine the contents of $\texttt{INCOMING}_t$ to determine if any transfers are scheduled to complete in the next cycle. This is accomplished with the clause marked **(1)** in Figure 4.4:

```
for each member I of INCOMING_t
    if deadline(I) == C + 1
```

The first of these lines can be rewritten as:

while $(\{\exists I | I \in \texttt{INCOMING}_t\})$

As there will be a finite number of members in $\texttt{INCOMING}_t$, this loop will terminate. As C is the current cycle, and deadline(I) is the cycle in which the transfer will complete, the check will identify the members of $\texttt{INCOMING}_t$ which we are interested in.

For the transfers which are going to complete in the next cycle, a check is made to determine if any of them are targeting a register which is currently PENDING. Any such registers are READ and removed from $\texttt{PENDING}_t$ by the clause marked **(2)** in Figure 4.4:

```
if target(I) ∈ PENDING_t
    issue READ of target(I)
    remove target(I) from PENDING_t
```

After this code has been executed it can be asserted that there are no remote transfers that will complete in the next cycle which target a register which is PENDING. It can also be asserted that $\texttt{PENDING}_t$ is correct up to this point.

For the transfers which are going to complete in this cycle, all that is necessary is that the description of the transfer be removed from $\texttt{PENDING}_t$. As the register allocator and scheduler are already aware that there is a transfer that is supposed to complete this cycle, no attempt will be made to allow another instruction to WRITE or LOAD to target(I). This is trivially accomplished by the clause marked **(3)** in Figure 4.4:

```
else if deadline(I) == C
    remove I from INCOMING_t
```

Following the execution of this clause, it can be asserted that $\texttt{INCOMING}_t$ is correct.

4.2 VLIW-Unknown

The next architecture to be considered departs froms the VLIW+ design described in Section 4.1 in two ways. The latency for remote register transfers is unknown at compile time, and the hardware that performs the remote data transfers does not suspend operation when the processors stall. This makes remote register transfers identical to issuing a LOAD operation with a remote destination.

This creates a scenario similar to that shown in Figure 4.2, but with the difference that there could be a processor stall inbetween steps **(3)** and **(4)**. Although the algorithm for managing this scenario may be less efficient than that for a known

transfer latency, it is simpler: prior to issuing the remote WRITE, a check is made of the register to determine its state. If it is PENDING, the register is READ before the WRITE is allowed to issue. This will ensure that the outstanding memory operation has completed. The register is then marked as PENDING. This solution provides the same latency-hiding potential for remote transfers as is provided for memory operations. It also contains the same potential for stalling due to an incomplete transfer as for an incomplete LOAD.

To implement this solution, no additional data need be maintained beyond what is required for the basic trace-scheduling algorithm shown in Section 1.6. The only adjustment is that remote transfers are treated identically to LOADs with a remote target. Both of these operations can be handled as described in Section 4.1. Prior to issuing the operation the register state of the target register is examined. If the target register is PENDING, a READ of it needs to be scheduled. Unless the lower-bound on the transfer latency is quite small (zero or one cycles), this READ can be scheduled in either the same cycle as the remote transfer, or the following cycle. A lower bound of zero cycles will be assumed to preserve generality. With a transfer latency of zero, it is necessary to issue the synchronizing READ of a register in either the same cycle as the remote transfer, or in the previous cycle.

4.3 MAP

The design for the MIT M-Machine includes a feature known as 'processor coupling' [34]. This is a hardware design idea that allows a chip to be constructed out of multiple clusters of functional units, each containing its own associated register file, with the ability to write values directly into the register files of the other clusters. Each cluster has its own Instruction Pointer (IP), and there is no guarantee made about the relative values of IPs on different clusters. While these clusters can be run

as autonomous units, the intended use is to have all of the clusters execute different portions of the same instruction stream, synchronizing when necessary to transfer data between clusters.

There are two major issues that this loose processor organization presents to the compiler. The first of these is managing register synchronization which is the focus of this thesis. The second issue is that of effective scheduling of instructions across loosely coupled processors. As is the case with most parallel programming, relative to computation, synchronization within such a system is expensive. The ability of a compiler to generate efficient code for a loosely-coupled architecture, and therefore the utility of such a design, is an open question. This is currently a topic of interest for both the Scalable Concurrent Programming Laboratory at Caltech and the Concurrent VLSI Architecture Group at MIT.

The major register synchronization issue presented by the MAP architecture is the sharing of register files among multiple asynchronous threads of control. In practice, this issue is simpler than it might appear. The asynchronous nature of the threads requires explicit synchronization before the transfer of data between threads can occur. Without explicit synchronization, a register could be overwritten before the receiving thread was able to `EMPTY` it. This guarantee of synchronization permits the use of the same algorithm proposed for a lockstepped machine with unknown transfer latency described in section 4.2.

Chapter 5 Compiler Development and Architectural Evaluation

This chapter describes some of the implementation details of the retargeting of the Multiflow compiler to the MIT M-Machine. These details are accompanied by commentary on, and critique of, the MAP instruction set architecture. The first part of this critique is an exploration of how the M-Machine architecture addresses criticisms [41, 40] leveled at its predecessor, the J-Machine [14].

The major complaints about the J-Machine architecture were the distinction between message buffer memory, and the lack of hardware floating-point support. Both of these concerns have been addressed in the M-Machine. The location of messages within memory is now under software control, and can be determined on a per-message basis. This allows the capability to copy message data out of the network directly into its final location. The M-Machine floating-point support is quite good. Each cluster of functional units has its own floating-point unit, for a total of three on each chip. In addition to standard floating-point arithmetic functions (+,-,*,/), the floating-point units also perform integer division (which was absent in the J-Machine), floating-point comparisons, and square root. All operations are done on a 64-bit IEEE floating-point format.

Other concerns raised about the J-Machine included the awkwardness of the memory addressing, the lack of byte addressing, the limitations on constant generation, and the lack of registers. The M-Machine solutions for all of these problems are significantly better than the solutions used in the J-Machine. Byte addressing has been implemented in a reasonable fashion, and the number of machine registers is

adequate. Memory addressing and constant generation are discussed later.

5.1 Predicated Operations

One of the features of the MAP instruction set is the ability to predicate any instruction on the value contained in a condition code (cc) register. This feature allows the compiler to generate very efficient code in situations where each clause in a conditional operation is very short. Two examples of how this is used within the compiler are `select` operations, including floating-point absolute value (FABS); and conditional branches.

5.1.1 Select Operations.

Select operations set a variable to one of two values based on a boolean expression. These operations are common in C code, and usually take one of the forms shown in Figure 5.1. Both of these examples can be compiled into very short instruction sequences as shown in Figure 5.2. It is clear from these examples that additional optimization could be performed to eliminate the copying of a register to itself.

---

(1) Conditional Using `If..Then`

```
if (a == b)
    c = a;
else
    c = 0;
```

(2) Conditional Using `?..:`

```
c = (a == b) ?
((a > 0) ? a : -a)
: 0;
```

Figure 5.1: **Conditional Expressions that can use Predicated Operations.** These two code fragments show common C language statements that can be efficiently compiled through the use of predicated operations.

---

(1) Conditional Using `If..Then`

```
ieq i6,i7, cc0;
ct cc0 mov i6, i6;
cf cc0 mov i0, i6;
```

(2) Conditional Using `?..:`

```
ilt i0,i6, cc0;
ct cc0 mov i6, i8;
cf cc0 mov i8, i8;
ieq i6,i7, cc0;
ct cc0 mov i8, i6;
cf cc0 mov i0, i6;
```

Figure 5.2: **Compilation of Conditionals using Predicated Operations**. This figure shows the assembly code output by the compiler for the examples shown in Figure 5.1.1.

FABS. A special case of a conditional which can be implemented using a predicated operation is floating-point absolute value (FABS). This is a function that is supported on both the SPARC-2 and the MIPS R4400 processors using an `fabs` instruction. The M-Machine compiler treats FABS as an instruction internally, but implements it using a predicated operation.

5.1.2 Conditional Branches

In addition to enabling the conditional filling of branch-delay slots, predicated operations provide a simple method for synchronizing registers at conditional changes in control flow. The one situation under which this capability is used is for register synchronization. At present, when a change in control flow is detected, and there are `PENDING` registers, these registers are synchronized prior to the change of control flow. The use of a conditional barrier in this instance prevents premature synchronization of registers if the branch is not taken.

5.2 64-Bit Execution

Unlike both the J-Machine, and all of the Multiflow Trace machines, the M-Machine has a word length of 64 bits. This change has a number of obvious advantages, including the ability to store floating-point values in integer registers, and an increased virtual address space. Such a fundamental change, however, required substantial changes within the compiler.

For example, it was necessary to rewrite every line which generated a 32-bit internal constant. This generation was originally performed with a call to a routine to generate an integer constant, with the assumption that integers were always 32-bit. These calls were replaced with a function which took a size argument to allow for generation of both 32- and 64-bit constants. One of the difficulties with this transition is that performing 32-bit initialization on a 64-bit constant leaves the high-order 32-bits of the constant undefined.

Another change was the necessity of creating 64-bit versions of most of the internal operations. For example, the M-Machine does not have a 32-bit integer addition, but does have a 64-bit integer addition. Although the compiler started out with a fairly complete set of opcodes, most of them had to be discarded in favor of 64-bit variants.

5.3 Local Register-to-Register Moves

Like many contemporary processors, each M-Machine cluster sub-divides its local register file into an integer bank and a floating-point bank. As was shown in Figure 1.3, all moves from one register bank to another, even if the other bank is in the same cluster, must go through the C-Switch. This introduces some variability to the latency of these moves, and requires some form of software intervention to assure proper synchronization. The latency of these moves is low enough that the compiler scheduling will avoid WAW hazards. What is necessary is for the functional unit

associated with the target register to issue an EMPTY instruction on the register before the move is initiated. As there is at least a 1-cycle latency for data to move across the C-Switch, the EMPTY can be issued in the same cycle as the move.

The additional instructions necessary for these EMPTY instructions, and the moves between integer and floating-point registers can compose a significant portion of instruction cycles during progam execution. This is particularly the case for programs that use a large amount of integer division and multiplication, as these operations require moving the operands over to the floating-point registers, and moving the result back to the integer bank. Table 5.1 shows the cost of EMPTY instructions as a percentage of executed instructions for unoptimized versions of the benchmarks described in Section 6.2. As these are operations whose latency can be bounded at compile-time, it is possible for the compiler scheduler to manage this piece of instruction choreography.

Program Name	EMPTY Percentage
MaMl	6.71
DIR	3.75
HASH	0.59
FFT	4.10
LU	4.79

Table 5.1: Count of EMPTY as a Percentage of Total Executed Instructions

5.4 Constant Generation

The technique for generating constants within the M-Machine is straightforward, but can be expensive for 64 bit constants. It is possible to generate 16 bits worth of constant with a single instruction, which mandates a cost of four instructions for a 64-bit constant. This requires some intelligence on the part of the compiler to avoid generating constants that are longer than necessary. For example, at compile-time

it is not possible to know how many bits are necessary to contain the address of a function. Early on in the M-Machine implementation the compiler naively assumed that all such constants were 64-bits in length. When it became apparent that constant generation accounted for a significant portion of instructions within a program, this was examined more closely. Although the M-Machine virtual address space is large enough that a 64-bit constant could be required to represent the address of a function, this is an unlikely scenario. Especially since the address generated is relative to a data pointer stored in a machine register. A decision was made that program code size was unlikely to grow beyond $2^{32} bytes = 4$ Gigabytes. Constant generation for function addresses was reduced to 32-bits.

A similar assumption was initially made for offsets from the stack pointer required for spilling and restoring registers. When this was examined more closely it became apparent that a more efficient solution could be utilized as these offset values are known at compile time. Unfortunately the uncertainty of whether one or two instructions will be needed does result in an inefficiency due to the structure of the compiler. In order to ensure proper choreography of instructions each operation is specified as taking a specific number of clock cycles to execute. The required machine resources for each of these cycles are specified. But in the case of a spill or restore we have created a situation where an operation can take a variable number of instructions. At present this requires that spills and restores insert a NOP instruction into the slot for the second constant generation instruction if a 16-bit constant is adequate. Future optimizations should be able to eliminate this spurious use of resources.

5.5 Memory Addressing

As already mentioned, the memory addressing used by the M-Machine allows the compiler to generate significantly better code than did the J-Machine addressing modes.

For accessing 64-bit data a straightforward load (`ld`) or store (`st`) instruction can be used. Address arithmetic is accomplished through the use of a load effective address (`LEA`) instruction which can either add to or subtract from the starting address.

Auto-increment Addressing Another method available for pointer arithmetic is an auto-increment feature available in both load and store instructions. These instructions take an optional operand which is an integer to be added to the pointer after the memory operation has been issued. Although the compiler does not yet take advantage of this feature, it is intended to support a very efficient way of implementing the pointer auto-increment feature of C. An example of how this could yield more efficient code is shown in Figure 5.5.

---

(1) Source Code

```
intcpy(a,b,size)
long *a, long *b, int size
{
    while(size--){
        *a++ = *b++;
    }
}
```

(2) Current Output

```
1: ialu ine i0,i8, cc0
   memu ld i6, i9;
2: ialu sub i4,#1, i4;
3: ialu cf cc0 br L1?3;
4: ialu lea i6,#8, i6;
5: memu mov i4, i8;
6: memu st i9, i7;
   ialu lea i7,#8, i7;
```

(3) Optimized Output

```
1: ialu ine i0,i8, cc0
   memu ld i6,#8, i9;
2: ialu sub i4,#1, i4;
3: ialu cf cc0 br L1?3;
4: memu mov i4, i8;
5: memu st i9, #8, i7
```

Figure 5.3: **Optimized Code Using Post-Increment Addressing**. This figure shows the source code for a trivial function, accompanied by the code currently generated by the compiler for the loop body, and the code that could be generated if the compiler took advantage of the post-increment addressing mode. The numbers to the left of the instructions represent the cycle in which each instruction takes place. The use of the post-increment addressing mode shortens the schedule for the loop within this routine by one cycle. It actually removes two instructions, leaving room for the compiler to move some other ialu instruction into cycle 5.

Byte Addressing The M-Machine memory operations are optimized for full-word operations. To support partial-word accesses instructions are provided to insert and extract bytes and half-words. All of these instructions require fetching the entire

word containing the desired data, and then performing the insert or extract. The insert and extract operations (`insb, insh, extb, exth`) take as arguments a register containing the data word to operate on, and a register either containing the data to insert, or to receive the extracted data. This second operand also contains the byte offset to be used for the insert/extract. One minor complication introduced by this structure is the necessity of maintaining both the data word retrieved by memory and the pointer to that word of memory.

5.6 Hardware Memory Segmentation

Part of the functionality of the M-Machine memory system is hardware memory segmentation and protection. Memory accesses can only be performed by dereferencing pointer data types. The ability to construct a new pointer (using the `setptr` instruction) is restricted to system code. User code can perform simple pointer arithmetic using the `LEA` instruction, as long as the new pointer obeys the segmentation limits of the old pointer. The layout of a pointer is shown in Figure 5.2. Segments are always 2^n words in length, with n specified in the segment length field of the pointer. The permission bits can be used to flag data as read only or read/write. There are a number of execute permissions that a pointer can have, including privileged and user.

Bits:	64	60-63	54 - 59	0 - 53
	Pointer Tag	Permission Bits	Segment Length	Address

Table 5.2: Structure of an M-Machine Pointer

Any attempt to access memory outside of a pointer's segment results in an error. This can cause difficulties in C programs that use common but imprecise memory accesses. One example of such an access can be found in the SPEC92 [54] benchmark program *espresso*. One portion of this program, shown in Figure 5.4, uses a macro

to traverse all elements within a dynamically created array. This loop terminates when the pointer to the current element of the list is equal to the element one past the end of the list. According to the C specification, it is legal to take the address of the first element past the end of an array [35]. Technically these elements are not within an array, however, but are aligned within a contiguous block of dynamically allocated memory. If the end of the array should fall on the segment boundary, taking the address of an element past the end of the array will generate an error on the M-Machine. Fortunately, this issue can be trivially addressed by taking the size requested for dynamic memory allocation and adding some extra space so that it is legal to address an item just past the end of the segment actually requested by the user program.

```
#define foreach_set(R, last, p)
    for(p=R→data,last=p+R→count*R→wsize;p<last;p+=R→wsize)
```

Figure 5.4: **Code Fragment from espresso.** This shows a macro definition used within the espresso benchmark from the SPEC Integer92 benchmark suite. This macro traverses a list of elements stored sequentially in a contiguous block of dynamically allocated memory. The termination condition of the loop is when the pointer to the current element is equal to the address of the first element past the end of the list.

Address Arithmetic and Pointer Copying. Two other areas in which the pointer structure of the MAP can create problems is performing arithmetic operations on pointers, and in performing block memory copies of memory that includes pointers. Both of these problems are encountered by the C library routine `memcpy()`. In it's simplest form this routine can be written as shown in Figure 5.5. The weakness in this implementation is that if the memory being copied contains pointers, only 64 bits of each pointer will be moved. The resultant memory will contain integer data rather than pointers as the pointer tag bit has been lost.

```
char *memcpy (char *ptoarg, char *pfromarg, long c)
{
    char *pto = (char *) ptoarg;
    const char *pfrom = (const char *) pfromarg;

    while (c-- != 0)
        *pto++ = *pfrom++;
    return ptoarg;
}
```

Figure 5.5: **Simple Implementation of memcpy**. This program shows a simple implementation of the C library routine `memcpy()`. As this routine performs copies one byte at a time, it will lose M-Machine data tags present beyond a word boundary. In particular, this copy will strip the pointer tag bit off of any pointer data items.

This problem with the loss of pointer bits suggests a different, and more efficient, implementation of `memcpy()`. This implementation, shown in Figure 5.6 attempts to word-align the source and destination pointers. It may be necessary to do a small number of byte copies for both winding up and winding down the copy, but the bulk of the copying is performed as word copies. This has the advantages of being more efficient than the simple implementation, and of ensuring that information, such as the MAP pointer tag bits, is not destroyed by the copy. If it is not possible to word-align the pointers, this routine defaults to using the byte copy shwon in Figure 5.5.

The drawback to the implementation shown in Figure 5.6 is that it performs arithmetic on MAP pointers. As these data items have information stored in high-order bits, they will frequently appear to arithmetic operations as negative numbers. This can easily result in incorrect results. To ensure correct results for arithmetic on pointers it is necessary to remove the high-order bits. This can be trivially accomplished with a pair of shift operations as shown in Figure 5.7.

```
char *memcpy (char *ptoarg, char *pfromarg, long c)
{
    char *pto = (char *) ptoarg;
    const char *pfrom = (const char *) pfromarg;
    int windin, windout;
    long *pf, *pt;

    windin = pfromarg % 8;
    windout = ptoarg % 8;

    if(windout && windin != windout) {
        while (c-- != 0)
            *pto++ = *pfrom++;
        return ptoarg;
    }

    c -= windin;
    while (windin-- != 0)
        *pto++ = *pfrom++;

    if(c < 0) {
        return ptoarg;
    }

    windin = c % 8;
    c -= windin;

    for(pf = (long *) pfrom, pt = (long *) pto ; c > 0 ; c -= 8)
        *pt++ = *pf++;

    pto = (char *) pt;
    pfrom = (char *) pf;

    while (windin-- != 0)
        *pto++ = *pfrom++;
    return ptoarg;
}
```

Figure 5.6: **Efficient Implementation of memcpy**. This program shows an efficient implementation of the C library routine `memcpy()`. This routine attempts to word-align the two pointers, and then do word copies of the memory. If it is not possible to word-align the two pointers, then this routine defaults to using the byte copy employed by the simple implementation shown in Figure 5.5

```
char *memcpy (char *ptoarg, char *pfromarg, long c)
{
    char *pto = (char *) ptoarg;
    const char *pfrom = (const char *) pfromarg;
    int windin, windout;
    long *pf, *pt;
    long ft, tt;

    ft = (int) pfromarg;
    ft <<= 10;
    ft >>= 10;
    tt = (int) ptoarg;
    tt <<= 10;
    tt >>= 10;

    windin = pfromarg % 8;
    windout = ptoarg % 8;
    :
}
```

Figure 5.7: **Correct Implementation of memcpy for the MAP Processor.** This program shows a change that must be made to the implementation shown in Figure 5.6 to ensure correct functioning on the MAP processor. If the high bits of a pointer are not removed prior to performing arithmetic operations on the pointer incorrect results can occur.

Chapter 6 Experimental Evaluation

This section describes a set of experiments and results that demonstrate the costs associated with WAW hazards on a selected set of benchmark programs[1]. Each program is characterized by the number of WAW hazards it experiences. Software WAW detection is compared to hardware WAW scoreboarding in terms of dynamic instruction counts and overall runtimes. In addition, the impact of compiler optimizations is discussed.

6.1 Experimental Environment

The experiments were conducted using executable programs generated by the M-Machine compiler, as well as an assembler and linker developed by members of the Concurrent VLSI Architecture Group at MIT. The programs were executed on `msim`, the simulator being used at MIT to perform architectural and logic validation of the M-Machine. The programs were all run on top of `mars` [26], the M-Machine runtime system, on one simulated M-Machine node. The user portion of the programs all ran within a single cluster (three functional units) of the M-Machine. Sixteen integer and sixteen floating point registers are available in the MAP chip. The programs were all compiled using `mmcc`, the Multiflow C compiler ported to the M-Machine, and enhanced with software WAW scoreboarding and detection.

For the purposes of these experiments `msim` was modified to support two different memory models, as shown in Table 6.1. `MM` is the memory system that will be used in the actual M-Machine hardware. This system uses a register scoreboard that only

[1]Some of the results from this chapter are also presented in [39].

synchronizes on READ accesses. SCOREBOARD is a memory system similar to that found in commodity processors such as the DEC Alpha 21164. This system uses a register scoreboard that will synchronize on both READ and WRITE accesses. Other than the ability to synchronize on WRITE accesses, the SCOREBOARD and MM models are identical.

Model Name	Description
MM	M-Machine Model
SCOREBOARD	Register Scoreboarding for WAW Prevention

Table 6.1: Memory Models Used For Experiments

Each program used in these experiments was compiled with four different sets of compiler options as shown in Table 6.2. For SCOREBOARD, which has hardware detection and prevention of WAW hazards, the NONE and OPT compilations flags are used. NONE uses neither compiler optimization nor software WAW detection, while OPT uses the standard optimizations in mmcc. For the MM model, the software WAW detection and prevention of the WAW and ALL models must be used. The use of optimization is investigated because although optimization can remove some number of compiler-generated WAW hazards from the output code, it also increases register pressure which can shorten the amount of latency that the algorithm can hide by delaying synchronization.

Option Name	Description	Model use
NONE	No optimizations or WAW Prevention	SCOREBOARD
OPT	Standard Optimizations Only	SCOREBOARD
WAW	WAW Prevention Only	MM
ALL	Both Standard Optimization and WAW Prevention	MM

Table 6.2: Compile-Time Options Used For Experiments

6.2 Basic Benchmark Programs

The programs that were selected for this evaluation are shown in Table 6.3. The first three of these programs are small codes with easily understood memory access patterns. The last two from the SPLASH [52] suite have more complicated memory access patterns, and they have been slightly modified to eliminate the synchronization that is only necessary for running in parallel. Table 6.4 shows the number of instructions contained in each program both with and without optimization, as well as the number of inserted synchronization operations that are placed into the program by the software WAW prevention algorithm. Only a small number of instructions need to be added even into the largest of the benchmark programs. In general, the use of optimization increases the number of instructions that need to be inserted.

Program Name	Description	Source	Problem Size
MaMl	Matrix Multiplication	SCP Lab	32 x 32 Matrix
DIR	Dirichlet Heat Transfer	SCP Lab	50 x 50 Grid
HASH	Hash Table Insert/Retrieve	SCP Lab	1000 Entries
FFT	Fast Fourier Transform	SPLASH Suite	1024 Doubles
LU	LU Decomposition	SPLASH Suite	32 x 32 Matrices

Table 6.3: Benchmarks Used For Experiments

`MaMl` is a matrix multiply that uses a simple triple-nested loop to read across rows of one matrix, and columns of the other. This represents the most trivial operation likely to be relevant to performance of scientific codes. After some initial compulsory misses, the cache hit rate is close to 100% as the cache can hold all of the elements of all of the matrices. The WAW hazards detected by the compiler occur only at branch points.

`DIR` is a program to solve LaPlace's equation $\nabla^2\theta = 0$ using constant boundary conditions and a Gauss-Seidel iteration scheme. This is intended to recreate the memory access pattern seen in the large-scale implicit/explicit Navier-Stokes applications

developed by the Scalable Concurrent Programming Laboratory [57, 56, 55]. The inner loop iterates over all elements in a 2-dimensional grid, and averages it with its four neighbors to the left, right, top, and bottom. As with `MaMl`, there will be occasional cache misses, but in general cache locality will provide immediate service of memory requests.

`HASH` is a hash table entry and retrieval program that is specifically designed to expose performance weaknesses of the memory system. This code is typical of that used in Particle-in-cell (PIC) or Direct Simulation Monte Carlo (DSMC) simulation techniques [50, 49, 51]. The bulk of this code dereferences pointers and traverses linked lists. The problem size is large enough that cache misses are likely. Code that can be optimized away will mask some of the latency of memory operations, however optimization opportunities are restricted as very little computation is performed between hash table accesses.

`FFT` performs a Fast Fourier Transform, which is frequently seen as the inner loop of many applications in image and signal processing. Most of the loops within this program are double-nested. In addition, the computation within most of the loops is very simple. The problem size is small enough to allow most of the data to reside in cache.

`LU` performs Lower-Upper Decomposition on a matrix, an operation frequently performed by linear algebra applications. One of the aspects of this program that complicates its memory behavior is that several of the nested loops within the program use a subroutine to perform the innermost loop. This results in increased register pressure within the loop, and increases the need for synchronization between iterations of the loop.

Program Name	Instructions	Memory Barriers	Register Barriers
Without Optimization (WAW)			
MaMl	1156	6	2
DIR	637	4	0
HASH	2446	25	13
FFT	8959	40	92
LU	6782	42	41
With Optimization (ALL)			
MaMl	1038	7	0
DIR	507	4	0
HASH	2231	35	19
FFT	8033	83	108
LU	5488	60	59

Table 6.4: Count of Inserted Barriers for Benchmark Programs

6.3 SCP Applications

In addition to the benchmark programs which were actually executed on the M-Machine simulator, a study was done of some of the large-scale applications codes in use within the Scalable Concurrent Programming Laboratory. This study measured the number of inserted barrier instructions required when compiling these applications for the M-Machine.

The applications examined are a Particle-In-Cell code (PIC) [50]; a three dimensional concurrent DSMC code (HAWK) [49, 51]; and a code which computes both steady state and unsteady solutions to the three-dimensional, compressible Navier-Stokes equations (ALSINS) [57]. Each of these programs was compiled both with and without optimizations.

The data for the inserted barriers required in these applications are shown in Table 6.5. In addition to the information gathered for Table 6.4, these table also indicates how many of the inserted barriers occur at branches. It is possible that the total number of barriers required for these programs could be reduced if the portion of the

SRS algorithm that optimizes transitions between traces were fully implemented.

All of the application programs require significantly more inserted barriers than do the smaller benchmark programs. For most of the programs the inserted barriers make up between 2% and 3% of the total number of instructions in generated code. The two exceptions are ALSINS without optimization where the barriers are less than 1% of the total instructions, and PIC with optimization where the barriers are slightly over 6%. Approximately 2% of these inserted barriers all occur within a single routine. This routine, **PhyPgetf**, manages the packing of data into messages to be sent to neighboring nodes. The bulk of this routine is a `switch..case` statement with 26 clauses all of which include a loop, and most of which have a function call within that loop. Another 1.4% of the inserted barriers comes from the code that handles removing data from incoming messages. This code has many of the same features requiring synchronization as **PhyPgetf**. As the vast bulk of these inserted barriers would not be encountered on any given execution of these functions, the totals appear worse than they are likely to be in terms of impact at runtime.

Program Name	Instructions	Memory Barriers	Register Barriers	Branches
Without Optimization (WAW)				
PIC	60,516	1238	350	907
HAWK	65,444	964	661	456
ALSINS	282,975	1004	945	619
With Optimization (ALL)				
PIC	116,017	4166	2878	1672
HAWK	63,073	1189	742	356
ALSINS	180,876	3005	1912	882

Table 6.5: Count of Inserted Barriers for SCP Applications

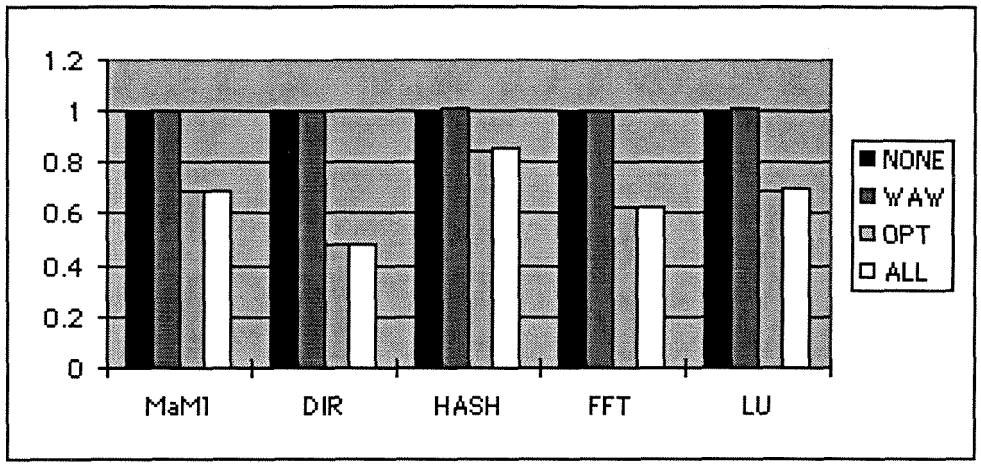


Figure 6.1: Dynamic Instruction Counts - Normalized to NONE

6.4 Dynamic Instruction Overhead

The *dynamic instruction overhead* shows the number of additional instructions required to implement WAW detection in software. This is a function of both the cost (in instructions) of each software WAW synchronization and their frequency. Figure 6.1 shows the dynamic instruction counts of both optimized and unoptimized models, normalized relative to the unoptimized code with no software WAW detection. As can be seen in this graph, there is essentially no cost in added instructions of using software WAW prevention. Programs that use optimization perform significantly better than those that do not.

The only programs that display a noticeable overhead from the software WAW detection are LU (1.78%) and HASH (1.48%). The overhead in LU is due to the cost of synchronization needed to manage the use of a subroutine call as an inner loop for much of the performed computation. The relevant piece of code is shown in Figure 6.2. This function requires memory synchronization before it is called, and increases register pressure within the loop.

```
for (k=0; k<n; k++) {
    for (j=k+1; j<n; j++) {
        a[k+j*stride] /= a[k+k*stride];
        alpha = -a[k+j*stride];
        length = n-k-1;
        daxpy(&a[k+1+j*stride], &a[k+1+k*stride], n-k-1, alpha);
    }
}
```

Figure 6.2: **Inner Loop of LU**. The call to the function **daxpy** in this loop accounts for a significant amount of the synchronization cost for the LU benchmark. The function call increases the already high level of register pressure within this routine.

The overhead incurred by HASH is the result of the code containing many branches and function calls. Due to the small amount of code in between changes in control flow, there is frequently a need for synchronization at these points in the program. This can be seen in the first portion of the HASH program, shown in Figure 6.3, which does all of the insertions into the hash table.

```
for(i = 0 ; i < ITERATIONS ; i++) {
    theKey = hash_value_to_key((void *) &seeds[i]);
    hash_add(&theTable, theKey, &theValue);
    *(int *)theValue = seeds[i];
    keys[i] = theKey;
}
```

Figure 6.3: **Hash Table Insertion from the HASH Benchmark** . The general structure of the HASH benchmark is a small number of memory references mixed in with calls to a hash-table management library. The small amount of code inbetween function calls makes it likely that there will be a need for synchronization prior to transferring control to the called function.

6.5 Performance Overhead

The total impact of software WAW detection can be determined by the number of cycles actually spent on each benchmark. This is a function not only of the cycle overhead of software WAW detection, but also the number of cycles spent waiting for the WAW to be resolved. Figures 6.4 and 6.5 show the breakdown of the number of cycles for each benchmark, into software WAW overhead cycles, execution cycles, time spent blocked on the memory system, and time spent waiting for WAWs to be resolved. For the SCOREBOARD model (denoted with SCORE), the three components of program execution time are instructions issued (INST), time stalled waiting for memory synchronization of RAW hazards (RAW), and time stalled waiting for memory synchronization of WAW hazards (WAW). The components for the MM model are INST, RAW, time stalled waiting for memory synchronization of WAW hazards (WAW), and overhead due to instructions issued to implement SRS (SRS).

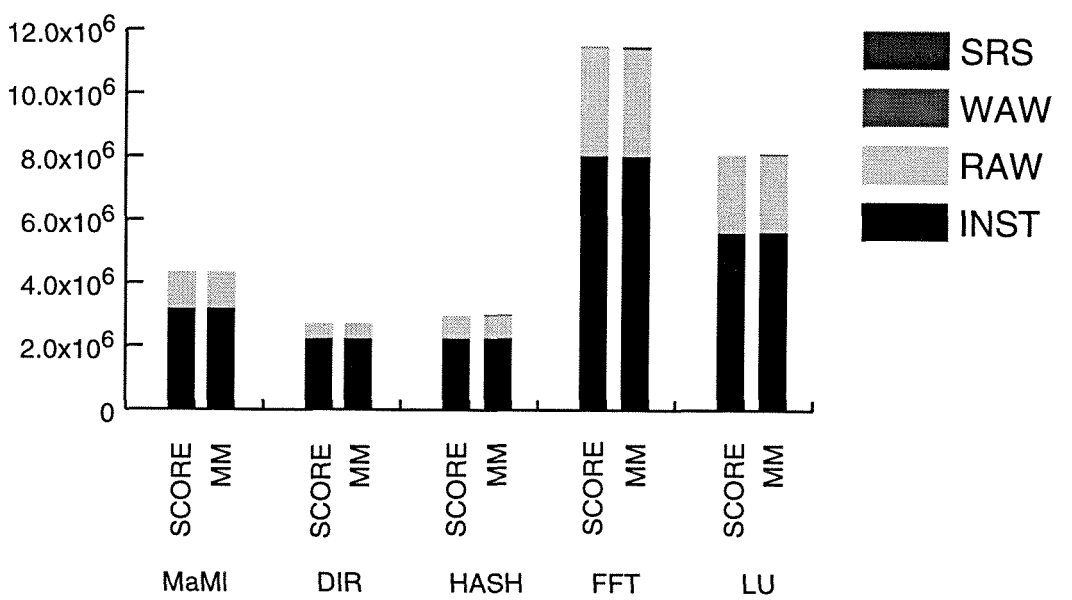


Figure 6.4: Hardware vs. Software WAW Prevention: No Optimization

On the SCOREBOARD model the largest component of program execution time is

actual instructions issued, followed by time spent awaiting memory synchronization due to RAW hazards. The only program that shows discernible cost awaiting resolution of WAW hazards is FFT, but this cost is negligible. Programs on the MM model spend a small amount of time managing WAW hazards, but this is a tiny portion of the total program execution time.

Figure 6.4 shows that the performance of software WAW prevention solution is comparable to that of the hardware solution. MaMl and DIR perform as well on the MM model as they do on the SCOREBOARD model. FFT and LU do not perform quite as well on the MM model, but the performance loss is quite small, with a peak performance difference of less than 1%. HASH is slightly worse as Software WAW detection is 2.3% worse than the hardware–only model.

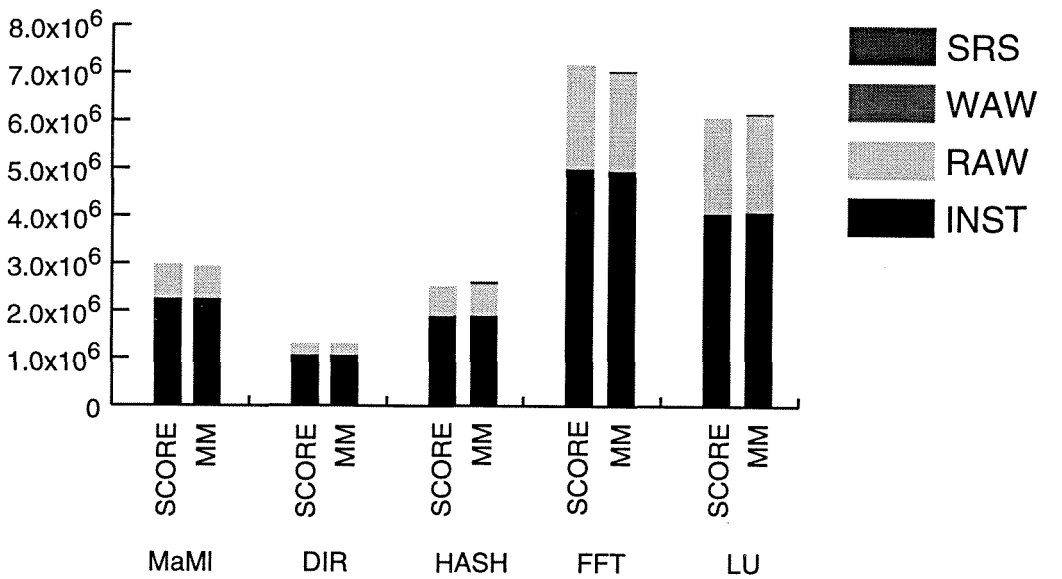


Figure 6.5: Hardware vs. Software WAW Prevention: Optimization

With optimizations, the simplest of the benchmarks in both program structure and memory access pattern were virtually identical both comparing the MM and SCOREBOARD models; and with and without compiler optimization. Because these programs, DIR and MaMl, are memory latency bound (ie. they have very little compu-

tation to hide memory latency), optimizations are not likely to improve the overlap between computation and memory access. The dominant effect of optimization in these programs is to reduce the total instruction count, without significantly affecting the number of WAWs.

The relative execution time of LU increases for software WAW detection as the result of optimization. LU has a significant amount of computation between memory references, which is reduced by optimization. This decreases the program's ability to cover memory latency with computation. Even with this additional degradation, the performance loss in LU for using software WAW prevention is less than 1%, still low enough to warrant this simplification of hardware.

Surprisingly, the execution time of the optimized FFT benchmark is actually less (almost 5% better) when using software WAW detection than when using the SCOREBOARD model. Since the program for the software WAW detection is virtually identical to that for SCOREBOARD, except for the additional synchronization instructions, one would expect the MM model to be strictly worse than SCOREBOARD. Part of the reason for the improvement of software WAW detection, is the register allocation biasing performed by the software solution. If the register biasing is used, but without inserting synchronizing operations, the gap between hardware and software shrinks to 2%. This remaining difference can be accounted for by examining the number of data cache misses incurred. The version of the program without the synchronizing operations encounters 76,552 more data cache misses than the program with inserted operations. This is an example of the change in program performance that can be caused by cache effects. This fluctuation is of the same order of magnitude as the performance degradation caused by using software WAW detection.

Finally, HASH gets significantly worse under optimization, showing a 6% increase in runtime over the hardware model. The HASH benchmark was designed to expose all of the latencies in memory accesses. The memory access pattern for this program is

irregular, and there is essentially no computation that can be overlapped with memory accesses. As virtually all of the memory accesses are within loops, there is a high likelihood that the software WAW prevention algorithm will force synchronization to occur earlier than the hardware algorithm, and thereby eliminate what little overlap of computation and memory access could potentially exist in the program.

6.6 Application Experiment

This section presents experimental results for an application program, the UNIX `compress` program. This program uses adaptive Lempel-Ziv encoding [62] to reduce the size of an input file. This program performs a large amount of I/O which is presently trapped by the simulator and executed on the host. The program performs computation to implement the compression algorithm, as well as hash table manipulations to keep track of intermediate data.

The count of inserted barriers for this program is shown in Table 6.6. These numbers are consistent with the numbers seen for the benchmark programs discussed in Section 6.2. The number of inserted instructions is a small part of the total number of instructions in the program, and optimization increases the number of instructions that are inserted.

	Instructions	Memory Barriers	Register Barriers	Branches
NONE	5385	61	21	10
WAW	5492	97	29	29

Table 6.6: Count of Inserted Barriers for Compress

Figure 6.6 shows the performance results from executing `compress`. The input file for these runs was a 100,000 byte reference file provided with the SPEC92 benchmark suite [54]. The compressed file produced by the program is 53,475 bytes long. The

relatively small change in the execution time between the optimized and unoptimized versions of this code is due to the significant portion of the program that is consumed by the overhead of reading in and writing out the data files. Even though much of this work is carried out by the simulator and the host, the program still incurs the overhead for making the calls to the I/O routines.

The overall result of these numbers is that there is negligible difference between the SRS algorithm and *register scoreboarding* for `compress`. SRS requires only 1% more execution cycles than does HRS, and SRS-OPT uses 0.13% more cycles than HRS-OPT. This supports the claim that the relatively poor performance of the `HASH` benchmark with SRS is due to the lack of computation between loop iterations. In comparison to `HASH`, there is a large amount of computation between calls to hash table routines in `compress`. While the use of different hash routines in the two programs precludes a direct comparison, the difference between the two results is compelling.

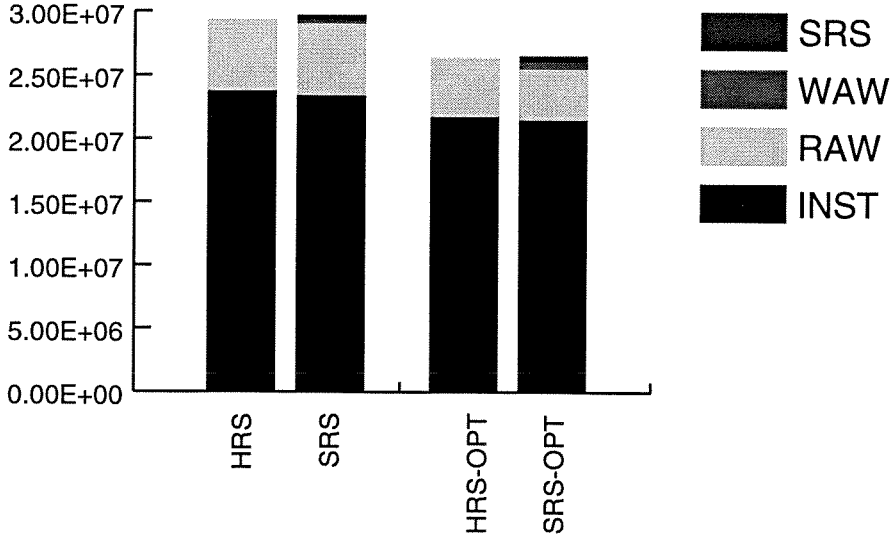


Figure 6.6: Hardware vs. Software WAW Prevention for Compress

6.7 Operating System Experiment

This section presents results for an application executed using the Scalable Concurrent Programming Library (SCPlib) [58] which is a library to support concurrent programming which exhibits operating system characteristics including preemptive multi-tasking and support for message-passing and I/O. The base application being executed on top of this environment is the Dirichlet problem described in Section 6.2.

SCPlib provides basic programming technology to support irregular applications on scalable concurrent hardware. This technology has been successfully applied to a variety of large-scale industrial application problems. The technology is based on the concept of a concurrent graph that provides an adaptive collection of light-weight threads that may relocate between computers dynamically. The graph is portable to a wide range of high-performance multicomputers (e.g. Cray T3D/E and Intel Paragon), shared-memory multiprocessors (e.g. SGI PowerChallenge), and networked workstations (e.g. IBM, SGI, or Sun workstations). For each architecture it is optimized to take advantage of the best available underlying communication and synchronization mechanisms.

The experiments described in this section were executed using a single-node version of SCPlib ported to the M-Machine. The use of this portable framework will allow for future direct comparisons between the M-Machine and existing commercial parallel computers. With the exception of load-balancing this single-node implementation utilizes all of the features of SCPlib. While it does not yet support internode communication, this is a special-case of multi-threading and inter-thread communication which are used for these experiments.

Table 6.7 shows the count of inserted barriers for SCPlib. These numbers are consistent with the numbers seen for both the benchmark programs discussed in Section 6.2, and the `compress` application discussed in Section 6.6. The statement made

about the results for `compress` still hold here: the number of inserted instructions is a small part of the total number of instructions in the program, and optimization increases the number of instructions that are inserted.

	Instructions	Memory Barriers	Register Barriers	Branches
NONE	66873	604	500	70
WAW	74820	1149	627	277

Table 6.7: Count of Inserted Barriers for SCPlib

Figure 6.7 shows the performance results from using SCPlib to execute a program calculating a solution to the Dirichlet problem on a 10×10 grid. Unlike previous experiments, this problem is run as a parallel program. The grid is partitioned among four processes, and SCPlib manages all of the process scheduling and inter-process communication. The execution of all four processes occurs on a single node of the MAP. Once again, the results show a negligible difference between the SRS algorithm and *register scoreboarding.* Without optimization, the difference between HRS and SRS is only 0.9%. While optimization does result in large increase in the difference, this increase still results in a small, 3.49%, performance penalty for use of SRS. This degradation in performance is due in part to changes in control flow requiring synchronization, which increase fourfold when using optimizations.

6.8 Discussion

In general, using a software algorithm to prevent WAW hazards is not significantly more expensive than preventing these hazards in hardware. This is due in part to the rarity of these hazards. In addition, the ability provided by the MAP of preventing a hazard using a single instruction reduces the cost of the software solution. In

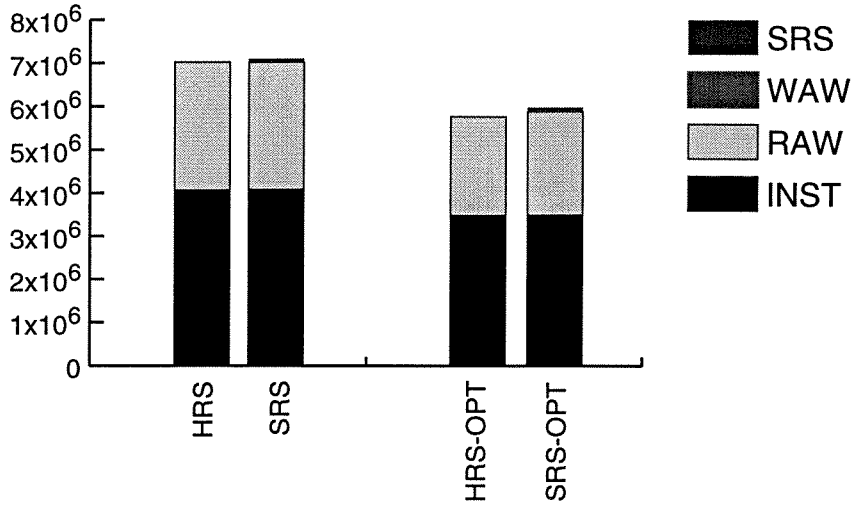


Figure 6.7: Hardware vs. Software WAW Prevention for SCPlib

addition, some amount of the runtime overhead incurred by the software algorithm would have been incurred by hardware WAW detection anyway. Other costs within the memory system, such as cache misses and synchronizing on RAW hazards, have a much greater impact on program performance than management of WAW hazards. The main observations from the experiments are summarized below:

- **WAW hazards occur infrequently.** As can be seen in Table 6.4 there are very few places within the benchmark programs where the compiler detected WAW hazards. Even in the worst case, the code inserted to manage WAW hazards comprises less than 2.4% of the program. Figures 6.4 and 6.5 show a very small cost for WAW hazard management in all programs on all models.
- **Software WAW prevention requires only one instruction per hazard.** As was discussed in Section 3.2, the MAP processor has a hardware memory barrier (`mbar`) instruction. In addition, a single register barrier can be implemented by performing a `READ` on the register. Both of these methods of synchronization require only one instruction to implement.
- **Software WAW detection has very little impact at runtime.** In the worst case, programs using software WAW detection incurred a 6% performance degradation relative to using hardware WAW detection, which is seen in a program designed to expose weaknesses in the memory system. In general, the software overheads are less than 2%.

- **Memory effects have greater impact on execution time.** As can be seen in Figures 6.4 and 6.5, the greatest cost in program execution other than issuing of instructions is waiting for memory synchronization following either cache misses or hardware detection of a RAW hazard. Relative to the time spent managing these other synchronization events, overhead needed for WAW detection and management is negligible.

Chapter 7 Conclusions

7.1 Results

In this thesis I have described how compiler support can eliminate certain pipeline hazards, thus alleviating the need for added hardware complexity. Read-after-write (RAW) hazards need some form of hardware support, especially when instructions, such as memory instructions in a lockup-free cache, may have variable latency. RAW synchronization is usually accomplished using pipeline stalling or register scoreboarding. Unlike RAW hazards, which are true data dependencies, write-after-read (WAR) and write-after-write (WAW) hazards are artifacts of a finite number of machine registers. The compiler algorithm for SRS (Software Register Synchronization) uses a compile-time scoreboard to detect when WAW hazards may occur. It then inserts additional instructions to convert these hazards to RAW hazards that can be handled by the simple hardware register scoreboard.

The experiments described in Section 6 compare WAW scoreboarding to SRS. In general, WAW scoreboarding performs at best only 2% better than SRS, as WAW hazards are actually rare events. The one exception to this performance was seen in `HASH`, a program specifically designed to expose weaknesses in the memory system. None of the programs spent a significant percentage of execution time managing SRS. This low cost can be largely attributed to the rarity of WAW hazards, combined with the ability to convert them into RAW hazards using a single instruction. The overhead incurred for RAW hazard management, even when using WAW scoreboarding, is a far more significant factor than efficient WAW hazard management in program performance.

7.2 Future Work

One of the most interesting open questions raised by this thesis is how well SRS will perform in a multi-cluster implementation. The complication in answering this question is that it requires a good multi-cluster implementation of the compiler, which does not yet exist. The design of such a compiler is in itself an interesting research topic.

It would be interesting to implement the full SRS algorithm to determine what effect that has on performance. While the algorithm must sometimes be more conservative than hardware, it should be capable of closing the gap between current performance and hardware.

While this thesis has demonstrated that SRS is a good substitute for hardware register scoreboarding, it is unknown how SRS performs relative to hardware register renaming.

7.3 Conclusion

Current high performance microprocessors use expensive hardware mechanisms to eliminate WAW hazards at runtime. This is due both to the desire to have object code compatibility, and to allow out of order instruction issue, as seen in some superscalar processors. The simplest mechanism is to use a scoreboard to track both register reads and register writes. Register renaming is an effective technique for eliminating WAW hazards at runtime, while also allowing out of order instruction issue. Both of these techniques require a table that must be accessed for all operands and destinations of every issuing instruction. Like the register file, the size of this table increases with the square of the number ports, making it large and slow as the number of arithmetic units increases.

As the number of function units increases on a single chip, register files will be

partitioned and access to them will be restricted to a small cluster of function units. Fast interaction latencies between clusters will be critical to high performance and the ability to exploit instruction level parallelism across the clusters. However, hardware WAW detection using scoreboarding or register renaming will become unattractive as maintaining global register state for use by every execution unit will be prohibitively slow and expensive.

Involving the compiler in preserving the order of instructions at runtime is a novel and necessary innovation to enable reduced hardware complexity. In the tradition of VLIW computers, this allows the compiler to perform better management of the hardware resources, including the registers. Machines may be made simpler and faster by removing complex hardware, and transferring those responsibilities to the compiler. As demonstrated by this thesis, WAW hazard detection can easily be performed in the register allocation phase of the compiler with little if any performance impact. If the clock rate can be increased at all, then it is likely to be a performance win. The strategy of making the compiler more intelligent will enable the continued increase in microprocessor performance as the improvement from those factors of the past (faster process technology and architectural innovations) become less significant.

Appendix A The MAP Instruction Set

This chapter provides a brief overview of the composition of the MAP instruction set. The structure of instructions is described, and a list of operations and their functionality is provided. This chapter lists some of the instructions that can be issued by user code, and does not list instructions that can only be issued by privileged code. Complete details on the MAP instruction set can be found in [15].

A.1 Anatomy of an Instruction

A MAP instruction consists of up to three operations, each of which are to be executed simultaneously on the same cluster of functional units. Each MAP chip is capable of executing three instructions (nine operations) at the same time. Within an instruction, each operation is designated by the functional unit on which it is to be executed.

An instruction is of the form:

instr	ialu	condition code	condition register	opcode	operands	
	falu	condition code	condition register	opcode	operands	
	memu	condition code	condition register	opcode	operands	;

Table A.1: Anatomy of an Instruction

The initial `instr`, and the terminal ; delimit the instruction. Each functional unit has one line to specify the operation to be executed on that unit. If any of these lines are omitted, no operation is scheduled on that functional unit. If all three lines are omitted, the instruction is considered to be a no operation (`NOP`). At execution time a `NOP` will consume an instruction issue slot, but will not perform any computation.

Conditional Execution: Each operation has optional fields for specifying a condition code and a condition register. If neither of these fields is specified, the default value is to always execute the operation. If a condition code and a condition register are specified, the operation is only executed if the condition code matches the boolean value in the condition register.

A.2 Listing of Operations

This section lists the possible values for the opcode field within an instruction, and describes the operands for each opcode. The following are conventions that are followed in these tables:

- Letters from a - c represent integers
- The letter d is a word whose lower 32-bits contain a datum to be inserted into a word, and whose upper 32-bits contain information as to where the datum is to be inserted.
- Letters from e - h represent floating-point numbers
- The letter i represent a 16-bit constant
- The letter s represents a scalar which could be either integer or floating-point
- The letters p and q represent pointers
- The letter r represents a bitmask of register IDs
- CC represent a condition code register
- IP represents the hardware Instruction Pointer

Opcode	Function	Op 0	Op 1	Op 2
add	c = a + b (signed)	a	b	c
sub	c = a - b (signed)	a	b	c
addu	c = a + b (unsigned)	a	b	c
subu	c = a - b (unsigned)	a	b	c
lea	q = p + a	p	a	q
leab	q = base of p + a	p	a	q
and	c = a AND b	a	b	c
or	c = a OR b	a	b	c
xor	c = a XOR b	a	b	c
ash	c = a SHIFT b (arithmetic)	a	b	c
lsh	c = a SHIFT b (logical)	a	b	c
rot	c = a ROTATE b	a	b	c
insb	c = Insert byte d into a	a	d	c
insh	c = Insert halfword d into a	a	d	c
extb	c = Extract byte b from a	a	b	c
exth	c = Extract halfword b from a	a	b	c
empty	Empty registers r	r		
ccempty	Empty condition registers r	r		
br	IP = IP + offset	offset		
jmp	IP = p	p		
imm	c = i	i	c	
shoru	b = a OR (b logical SHIFT 16)	a	b	
not	b = NOT a	a	b	
mov	b = a	a	b	
ilt	CC = a < b (signed)	a	b	CC
ile	CC = a <= b (signed)	a	b	CC
ult	CC = a < b (unsigned)	a	b	CC
ule	CC = a <= b (unsigned)	a	b	CC
ine	CC = a != b	a	b	CC
ieq	CC = a == b	a	b	CC

Table A.2: IALU Instructions

Opcode	Function	Op 0	Op 1	Op 2	Op3
fadd	h = f + g	f	g	h	
fsub	h = f - g	f	g	h	
fmul	h = f * g	f	g	h	
fdiv	h = f / g	f	g	h	
imul	c = a * b	a	b	c	
idiv	c = a / b	a	b	c	
fmula	e = (f * g) + h	f	g	h	e
fsqrt	g = $\sqrt{f}$	f	g		
fempty	Empty registers r	r			
fimm	h = i	i	h		
fshoru	g = f OR (g logical SHIFT 16)	f	g		
mov	g = f	f	g		
flt	CC = f < g (signed)	f	g	CC	
fle	CC = f <= g (signed)	f	g	CC	
fne	CC = f != g	f	g	CC	
feq	CC = f == g	f	g	CC	
itof	f = (double) a	a	f		
ftoi	a = (long) f	f	a		

Table A.3: FALU Instructions

Opcode	Function	Op 0	Op 1	Op 2
ld	s = contents of memory location p	p	s	
st	contents of memory location p = a	a	p	
fst	contents of memory location p = f	f	p	
mbar	Memory Barrier			
lea	q = p + a	p	a	q
leab	q = base of p + a	p	a	q
not	b = NOT a	a	b	
mov	b = a	a	b	
add	c = a + b (signed)	a	b	c
sub	c = a - b (signed)	a	b	c
and	c = a AND b	a	b	c
or	c = a OR b	a	b	c
xor	c = a XOR b	a	b	c

Table A.4: MEMU Instructions

Bibliography

[1] ANDERSON, D., SPARACIO, F., AND TOMASULO, R. The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling. *IBM Journal of Research and Development* (January 1967), 8–24.

[2] BERNSTEIN, D., GOLDIN, D., GOLUMBIC, M., KRAWCZYK, H., MANSOUR, Y., NAHSHON, I., AND PINTER, R. Spill code minimization techniques for optimizing compilers. In *Proceedings of ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation* (June 1989), ACM, pp. 258–263.

[3] BRADLEE, D. G., EGGERS, S. J., AND HENRY, R. Integrating register allocation and instruction scheduling for RISCs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (April 1991), ACM, pp. 122–131.

[4] BRIGGS, P., COOPER, K., KENNEDY, K., AND TORCZON, L. Coloring heuristics for register allocation. In *Proceedings of ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation* (June 1989), ACM, pp. 275–284.

[5] BRIGGS, P., COOPER, K., AND TORCZON, L. Aggressive live range splitting. Department of computer science technical report, Rice University, 1991.

[6] BRIGGS, P., COOPER, K., AND TORCZON, L. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems 16*, 3 (May 1994), 428–455.

[7] CALLAHAN, D., AND KOBLENZ, B. Register allocation via hierarchical graph coloring. In *Proceedings of ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (June 1991), ACM, pp. 192–203.

[8] CHAITIN, G. Register allocation and spilling via graph coloring. In *Proceedings of ACM SIGPLAN 1982 Symposium on Compiler Construction* (June 1982), ACM, pp. 98–105.

[9] CHAITIN, G. J., AUSLANDER, M., CHANDRA, A., COCKE, J., HOPKINS, M., AND MARKSTEIN, P. Register allocation via coloring. *Computer Languages 6*, 1 (January 1981), 47–57.

[10] CHARLESWORTH, A. E. An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family. *Computer 14*, 9 (September 1981), 18–27.

[11] CHOW, F., AND HENNESSY, J. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems 12*, 4 (October 1990), 501–536.

[12] COLWELL, R. P., NIX, R. P., O'DONNELL, J. J., PAPWORTH, D. B., AND RODMAN, P. K. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on Computers 37*, 8 (August 1988), 967–979.

[13] COLWELL, R. P., AND STECK, R. L. A 0.6um BiCMOS processor with dynamic execution. In *ISSCC95* (February 1995), IEEE, pp. 176–177.

[14] DALLY, W. J., ET AL. The J-Machine: A fine-grain concurrent computer. *Information Processing 89* (1989).

[15] DALLY, W. J., KECKLER, S. W., CARTER, N., CHANG, A., FILLO, M., AND LEE, W. S. The MAP instruction set reference manual v1.51. Concurrent VLSI Architecture Memo 59, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, August 1996.

[16] EDMONDSON, JOHN, H., ET AL. Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technical Journal 7*, 1 (July 1995).

[17] ELLIS, J. R. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, 1986.

[18] FARRENS, M. K., AND PLESZKUN, A. R. Implementation of the PIPE processor. *Computer 24*, 1 (1991).

[19] FILLO, M., KECKLER, S. W., DALLY, W. J., CARTER, N. P., CHANG, A., GUREVICH, Y., AND LEE, W. S. The M-Machine Multicomputer. In *Proceedings of the 28th International Symposium on Microarchitecture* (Ann Arbor, MI, December 1995), ACM, pp. 146–156.

[20] FISHER, J. A. Very long instruction word architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture* (1983).

[21] FISHER, J. A. Global code generation for instruction-level parallelism: Trace scheduling-2. In *Proceedings of the Workshop on Advanced Compilation Techniques for Novel Machine Architecture* (1991), Springer-Verlag.

[22] FISHER, J. A., AND FREUDENBERGER, S. M. Predicting conditional jump directions from previous runs of a program. Hewlett-packard laboratories technical report, Hewlett-Packard Laboratories, 1992.

[23] FREUDENBERGER, S. M., AND RUTTENBERG, J. C. Phase ordering of register allocation and instruction scheduling. In *Code Generation - Concepts, Tools, Techniques* (1991).

[24] GEORGE, L., AND APPEL, A. W. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems 18*, 3 (May 1996), 300–324.

[25] GIBBONS, P., AND MUCHNIK, S. S. Efficient instruction scheduling for a pipelined processor. In *Proceedings of ACM SIGPLAN 1986 Symposium on Compiler Construction* (June 1986), ACM, pp. 11–16.

[26] GUREVICH, Y. The M-Machine operating system. Master of Engineering Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, September 1995.

[27] GWENNAP, L. PA-8000 combines complexity and speed. *Microprocessor Report 8*, 15 (November 1994).

[28] HALL, M. W. Managing interprocedural optimization. Ph.D. Thesis, Rice University, Department of Computer Science, May 1991.

[29] HALL, M. W., MELLOR-CRUMMEY, J., CARLE, A., AND RODRIGUEZ, R. FIAT: A framework for interprocedural analysis and transformation. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing* (August 1993).

[30] HENNESSY, J. L., AND A., P. D. *Computer Architecture a Quantitative Approach.* Morgan Kaufmann, San Mateo, 1990.

[31] HENNESSY, J. L., AND GROSS, T. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems 5*, 3 (July 1983), 422–448.

[32] HENNESSY, J. L., JOUPPI, N., AND BASKETT, F. MIPS: A VLSI processor architecture. In *Proceedings of CMU Conference on VLSI Systems and Computation* (October 1981), Computer Science Press, pp. 337–346.

[33] HWU, W.-M. W., MAHLKE, S., CHEN, W. Y., CHANG, P. P., WARTER, N. J., BRINGMANN, R. A., OUELLETTE, R. G., HANK, R. E., KIYOHARA, T., HAAB, G. E., HOLM, J. G., AND LAVERY, D. M. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing 7*, 1-2 (May 1993), 229–248.

[34] KECKLER, S. A coupled multi-alu processing node for a highly parallel computer. Master of Science Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1992.

[35] KERNIGHAN, B. W., AND RITCHIE, D. M. *The C Programming Language, 2nd Edition.* Prenitce Hall, 1988.

[36] LAM, M. S. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (June 1988), ACM, pp. 318–328.

[37] LOWNEY, P. G., FREUDENBERGER, S. G., KARZES, T. J., LICHTENSTEIN, W. D., NIX, R. P., O'DONNELL, J. S., AND RUTTENBERG, J. C. The Multiflow trace scheduling compiler. *The Journal of Supercomputing 7*, 1-2 (May 1993), 51–142.

[38] MAHLKE, S. A., LIN, D. C., CHEN, W. Y., HANK, R. E., AND A., B. R. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture* (December 1992), pp. 45–54.

[39] MASKIT, D., KECKLER, STEPHEN, W., AND TAYLOR, S. Software Register Synchronization for Partitioned Register Files. *IEEE Transactions on Computers* (Submitted November 1996).

[40] MASKIT, D., AND TAYLOR, S. Experiences in programming the J-Machine. Department of Computer Science Technical Report CS-TR-93-11, California Institute of Technology, 1993.

[41] MASKIT, D., AND TAYLOR, S. A Message-driven Programming System for Fine-grain Multicomputers. *Software - Practice and Experience 24*, 10 (October 1994), 953–980.

[42] MEAD, C., AND CONWAY, L. *Introduction to VLSI Systems.* Addison Wesley, Menlo Park, California, 1980.

[43] MIPS TECHNOLOGIES, INCORPORATED. *R10000 Microprocessor Product Overview.* Sunnyvale, CA, 1994.

[44] MOTOROLA CORPORATION. *PowerPC 604 RISC Microprocessor Users Manual*, 1995.

[45] OEHLER, R., AND GROVES, R. IBM RISC System-6000 processor architecture. *IBM Journal of Research and Development 34*, 1 (January 1990), 23–36.

[46] PATTERSON, D. A., AND DITZEL, D. R. The case for the reduced instruction set computer. *Computer Architecture News 8* (October 1980).

[47] PATTERSON, D. A., AND SEQUIN, C. RISC-I: A reduced instruction set VLSI computer. In *Proceedings of the 8th Annual International Symposium on Computer Architecture* (1981).

[48] RADIN, G. The 801 minicomputer. *SIGARCH Computer Architecture News 10* (March 1982), 39–47.

[49] RIEFFEL, M. Concurrent simulation of plasma reactors for VLSI plasma manufacturing. Master of Science Thesis, California Institute of Technology, Department of Computer Science, 1995.

[50] ROY, S., HASTINGS, D., AND TAYLOR, S. Three-dimensional plasma particle-in-cell calculations of ion thruster backflow contamination. In *Proceedings of the 34th AIAA Aerospace Sciences Meeting* (1996), AIAA.

[51] SHANKAR, S., RIEFFEL, M., TAYLOR, S., JERDE, L., AND DITIZIO, R. Three-dimensional flow simulations in low pressure etch reactors. In *Invited talk for the 22nd Tegal Plasma Symposium* (1996).

[52] SINGH, J. P., WEBER, W.-D., AND GUPTA, A. SPLASH: Stanford parallel applications for shared-memory. Tech. Rep. CSL-TR-91-469, Stanford University, Apr. 1991.

[53] SITES, R. Instruction ordering for the CRAY-1 computer. Department of Computer Science Technical Report 78-CS-023, University of California, San Diego, 1979.

[54] SPEC benchmark release 1.1, 1992.

[55] TAYLOR, S., AND WANG, J. A concurrent Navier-Stokes solver for implicit multibody calculations. In *Proceedings of Computational Fluid Dynamics '93* (1993), Elsevier Science Publishers B.V.

[56] TAYLOR, S., AND WANG, J. A concurrent, nodal mismatched, implicit Navier-Stokes solver. In *Proceedings of Computational Fluid Dynamics '94* (1994), Elsevier Science Publishers B.V.

[57] TAYLOR, S., AND WANG, J. Launch-vehicle simulations using a concurrent, implicit Navier-Stokes solver. *Journal of Spacecraft and Rockets 33*, 5 (September-October 1996), 601–606.

[58] TAYLOR, S., WATTS, J., RIEFFEL, M., AND PALMER, M. The concurrent graph: Basic technology for irregular problems. *IEEE Parallel and Distributed Technology 4*, 2 (Summer 1996), 15–25.

[59] THORNTON, J. E. Parallel operation in the Control Data 6600. In *Proceedings of the Fall Joint Computer Conference* (January 1964), pp. 33–40.

[60] TOMASULO, R. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development 11*, 1 (January 1967), 25–33.

[61] TRIOLET, R., IRIGOIN, F., AND FEAUTRIER, P. Direct parallelization of call statements. In *Proceedings of ACM SIGPLAN 1986 Symposium on Compiler Construction, SIGPLAN Notices 21(7)* (July 1986), ACM, pp. 176–185.

[62] WELCH, T. A. A technique for high performance data compression. *IEEE Computer 17*, 6 (June 1984), 8–19.