Designing a Compiler for a Distributed Memory Parallel Computing System

by

Sidney Page Bennett

Thesis submitted to the faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Engineering

Dr. James M. Baker, Chairman

Dr. Nathaniel J. Davis

Dr. James D. Arthur

November 2003

Blacksburg, Virginia

Keywords: Compiler Design, SUIF, MachSUIF,
Optimization, Parallel Computing, Multithreading

Copyright 2003, Sidney Bennett

Designing a Compiler for a Distributed Memory Parallel Computing System

Sidney Bennett

James M. Baker, PhD, committee chair

Department of Electrical and Computer Engineering

Abstract

The SCMP processor presents a unique approach to processor design: integrating multiple processors, a network, and memory onto a single chip. The benefits to this design include a reduction in overhead incurred by synchronization, communication, and memory accesses. To properly determine its effectiveness, the SCMP architecture must be exercised under a wide variety of workloads, creating the need for a variety of applications. A compiler can relieve the time spent developing these applications by allowing the use of languages such as C and Fortran. However, compiler development is a research area in its own right, requiring extensive knowledge of the architecture to make good use of its resources.

This thesis presents the design and implementation of a compiler for the SCMP architecture. The thesis includes an in-depth analysis of SCMP and the necessary design choices for an effective compiler using the SUIF and MachSUIF toolsets. Two optimizations passes are included in the discussion: partial redundancy elimination and instruction scheduling. While these optimizations are not specific to parallel computing, architectural considerations must still be made to properly implement the algorithms within the SCMP compiler. These optimizations yield an overall reduction in execution time of 15-36%.

Contents

LIST OF FIGURES V

LIST OF EQUATIONS VII

LIST OF TABLES VIII

CHAPTER 1. INTRODUCTION 1

1.1 CURRENT PROCESSOR LIMITATIONS 1
1.2 AN SCMP OVERVIEW 2
1.3 COMPILER JUSTIFICATION 3
1.4 THESIS ORGANIZATION 3

CHAPTER 2. THE SCMP ARCHITECTURE 5

2.1 THE NODE 5
2.2 THE NETWORK 9
2.3 EXTERNAL COMMUNICATION 10

CHAPTER 3. A COMPILER OVERVIEW 12

3.1 SUIF INTRODUCTION 12
3.2 MACHSUIF INTRODUCTION 14
3.3 SUIF AND GCC 15
3.4 THE NEED FOR OPTIMIZERS 17
3.5 COMPILER SUMMARY 17

CHAPTER 4. THE SCMP BACKEND 18

4.1 MACHSUIF BACKEND PASSES. 18
4.2 CALLING CONVENTION AND REGISTER USAGE 19
4.3 REMOTE PROCEDURE CALLS. 23
4.4 REGISTER ASSIGNMENTS 29
4.5 STACK FRAME LAYOUT 30
4.6 MACRO INSTRUCTIONS 31
4.7 SCMP BACKEND SUMMARY 34

CHAPTER 5. PARTIAL REDUNDANCY ELIMINATION 36

5.1 LAZY CODE MOTION 38
5.2 CONTROL FLOW GRAPH PROPERTIES 39
5.3 EXPRESSION NAMING 44
5.4 MULTIPLE INSTRUCTIONS WITHIN A BASIC BLOCK 46
5.5 LOCAL DATA FLOW ANALYSIS 49
5.6 MOVING AND REMOVING MEMORY ACCESS EXPRESSIONS. 51
5.7 PARTIAL REDUNDANCY CONCLUSIONS 55

CHAPTER 6. OPTIMIZING WITH STATIC SINGLE ASSIGNMENTS 57

6.1 SSA INTRODUCTION. 57
6.2 PARTIAL REDUNDANCY USING THE SSA FORM. 58
6.3 SSA CONCLUSIONS 62

CHAPTER 7. INSTRUCTION SCHEDULING 64

7.1 THE SCMP PIPELINE 65
7.2 THE SCHEDULER ALGORITHM 67
7.3 SCHEDULER LIMITATIONS 78
7.4 SCHEDULER CONCLUSIONS 81

CHAPTER 8. TESTING AND RESULTS **82**

8.1 SIMULATION METHODOLOGY 82
8.2 RESULTS 83
8.3 OPTIMIZATION CHARACTERISTICS 87

CHAPTER 9. CONCLUSIONS **88**

9.1 FUTURE WORK 88
9.2 SUMMARY 89

BIBLIOGRAPHY **91**

APPENDIX A. THE SUIFVM INSTRUCTION SET **94**

APPENDIX B. SCMP RPC LIBRARY **96**

List of Figures

FIGURE 2-1. A SINGLE CHIP MESSAGE PASSING (SCMP) NETWORK. 5
FIGURE 2-2. PRIMARY COMPONENTS OF AN SCMP NODE. 6
FIGURE 2-3. THE ROUND ROBIN SCHEDULE OF SCMP CONTEXTS. WHEN THE THREAD IN CONTEXT 12 YIELDS THE PROCESSOR, THE SCHEDULER WILL SELECT CONTEXT 14 AS THE NEXT CONTEXT TO BE RUN. 9
FIGURE 3-1. SUIF TRANSFORMATIONS REQUIRED TO PREPARE A SUIF INTERMEDIATE FILE FOR MACHSUIF. 13
FIGURE 3-2. MACHSUIF TRANSFORMATIONS REQUIRED TO PRODUCE ASSEMBLY CODE 14
FIGURE 4-1. MACHSUIF COMPILER PASSES. PASSES INVOLVING THE SCMP BACKED ARE SHOWN IN BOLD. 18
FIGURE 4-2. SUIFVM INSTRUCTIONS AND THEIR SCMP EQUIVALENTS 18
FIGURE 4-3. SUIFVM INSTRUCTIONS REQUIRING MULTIPLE SCMP INSTRUCTIONS 19
FIGURE 4-4. EFFECTIVE TRANSFORMATION OF A PROCEDURE RETURNING A STRUCTURE 23
FIGURE 4-5. SCMP INSTRUCTIONS TO GENERATE A REMOTE PROCEDURE CALL. 24
FIGURE 4-6. SCMP STACK ORGANIZATION, WITH THE CALCULATIONS FOR LOCATING THE STACK SEGMENT FOR CONTEXT ELEVEN. 26
FIGURE 4-7. INSTRUCTIONS FOR CALLING A SCMP CALLBACK FUNCTION 27
FIGURE 4-8. AN SCMP REMOTE PROCEDURE 28
FIGURE 4-9. DEFINING AND CALLING A REMOTE PROCEDURE 28
FIGURE 4-10. STACK ORGANIZATION FOR AN SCMP PROCEDURE CALL. 31
FIGURE 5-1. AN EXAMPLE OF A PARTIAL REDUNDANCY ELIMINATION. 36
FIGURE 5-2. A SAMPLE CONTROL FLOW GRAPH. 40
FIGURE 5-3. OPTIMIZATION LIMITED BY A CRITICAL EDGE. 40
FIGURE 5-4. TRANSFORMED GRAPH PERMITS CODE MOTION. 41
FIGURE 5-5. A CRITICAL NODE PREVENTING CODE MOTION. 41
FIGURE 5-6. THE CRITICAL NODE PREVENTS ANY CODE MOTION 42
FIGURE 5-7. CREATING CRITICAL EDGES INSTEAD OF CRITICAL NODES ALLOWS CODE MOTION. 43
FIGURE 5-8. CRITICAL NODES CAUSED BY TEST AND BRANCH TRANSLATION. 43
FIGURE 5-9. LAZY CODE MOTION INCORRECTLY REMOVING A REQUIRED EXPRESSION. 44
FIGURE 5-10. LAZY CODE MOTION MOVING AN EXPRESSION CORRECTLY. 45
FIGURE 5-11. PARTIAL REDUNDANCY ELIMINATION WITH MULTIPLE INSTRUCTIONS IN A NODE. 46
FIGURE 5-12. INEFFECTIVE PARTIAL REDUNDANCY WITH MULTIPLE INSTRUCTIONS IN A BLOCK. 47
FIGURE 5-13. REDUNDANT EXPRESSIONS CAN'T BE PLACED ONLY AT THE ENTRY AND EXIT OF A BASIC BLOCK. 47
FIGURE 5-14. INVALID MOVEMENT OF AN EXPRESSION ACROSS A BASIC BLOCK BOUNDARY 48
FIGURE 5-15. LAZY CODE MOTION AT WORK WHERE PERHAPS IT SHOULDN'T BE. 49
FIGURE 5-16. PARTIAL REDUNDANCY IS NOT ABLE TO REMOVE REDUNDANT EXPRESSIONS UNTIL THEY RECEIVE THE SAME NAME. ONE PASS OF THE ALGORITHM MAY NOT BE SUFFICIENT. 50
FIGURE 5-17. TO REMOVE ALL POSSIBLE REDUNDANCIES, WE MUST APPLY PARTIAL REDUNDANCY ELIMINATION UNTIL NO CHANGE OCCURS. 51
FIGURE 5-18. POTENTIAL REDUNDANCIES IN MEMORY ACCESS EXPRESSIONS. 51
FIGURE 5-19. COMPLETE PROGRAM USING CODE OF FIGURE 5-1. 52
FIGURE 5-20. CODE SEGMENT OF A MULTITHREADED APPLICATION 53
FIGURE 5-21. A CORRUPTED MULTITHREADED APPLICATION. 53
FIGURE 5-22. INCORRECT REDUNDANCY ELIMINATION ON MEMORY ACCESS EXPRESSIONS. NOTICE MOVEMENT OF EXPRESSION A FROM THE LOOP. 54
FIGURE 5-23. PROCEDURES THAT ALLOW ACCESS BY MULTIPLE VARIABLES TO A COMMON MEMORY LOCATION. MEMORY ACCESS OPTIMIZATIONS MAY NOT BE POSSIBLE WITH PARTIAL REDUNDANCY ELIMINATION. 55
FIGURE 6-1. A CONTROL FLOW GRAPH AND ITS SSA EQUIVALENT 57
FIGURE 6-2. SSA GRAPH NEEDING A PHI NODE. 58
FIGURE 6-3. A CONTROL FLOW GRAPH WITH MULTIPLE ASSIGNMENTS TO A VARIABLE. 59

Figure 6-4. Partial redundancy incorrectly applied to figure 6-3. 59
Figure 6-5. Insertion of PHI instructions during partial redundancy elimination 60
Figure 6-6. Proper transformation of Figure 6-3 to eliminate partial redundancies 60
Figure 6-7. An invalid SSA graph generated due to a volatile variable. 61
Figure 6-8. A sample control flow graph with no memory access expressions and its SSA equivalent 62
Figure 6-9. The invalid SSA graph in Figure 6-8 generated after expression naming and redundancy elimination 62
Figure 6-10. PHI instruction added by partial redundancy elimination to maintain SSA validity 62
Figure 7-1. An instruction sequence causing a pipeline stall. 64
Figure 7-2. Reordering instructions to avoid pipeline a stall. 64
Figure 7-3. The SCMP pipeline 65
Figure 7-4. Examples of dependencies handled by the SCMP pipeline. 66
Figure 7-5. Pipeline stall on a memory read access. 66
Figure 7-6. Reorganizing loads to avoid pipeline stalls. 67
Figure 7-7. Several orderings resulting from a partial ordering. 68
Figure 7-8. Successive reads must be dependent on previous and successive writes 69
Figure 7-9. The memory writes dependent due to modified unknown base register value. 70
Figure 7-10. The two writes are independent regardless of the value of r5. 70
Figure 7-11. Pairs of writes that show dependencies due to mixed access types. 70
Figure 7-12. Invalid memory accesses that need not be considered. 71
Figure 7-13. Several memory accesses and the resulting dependency tree. 72
Figure 7-14. A sample program to be scheduled. 73
Figure 7-15. A Dependency tree of the program in Figure 7-14. 74
Figure 7-16. A list of ordered instructions in the ready set. 77
Figure 7-17. Affects of register allocation on code scheduling. 78
Figure 7-18. Register allocator introducing stalls after scheduling. Stall are not indicated until after scheduling. 79
Figure 7-19. Scheduler avoiding stalls after register allocation. Stalls are not indicated until after scheduling. 79
Figure 7-20. Movement of volatile loads may not be safe. 80
Figure 8-1. IFFT execution times as compared to baseline. 83
Figure 8-2. Median Filter execution times as compared to baseline. 84
Figure 8-3. Matrix Multiply execution times as compared to baseline. 84
Figure 8-4. IFFT simulation results with alias analysis as compared to the baseline. 85
Figure 8-5. Median filtering results with alias analysis as compared to the baseline. 86
Figure 8-6. Matrix Multiply results with alias analysis as compared to the baseline. 86
Figure 8-7. Aver age partial redundancy elimination performance of three application kernels. 87

List of Equations

EQUATION 5-1. DATA FLOW EQUATIONS FOR GLOBAL ANTICIPATIBILITY. 38
EQUATION 5-2. DATA FLOW EQUATIONS FOR EARLIESTNESS.. 39
EQUATION 5-3. DATA FLOW EQUATIONS FOR DELAYEDNESS... 39
EQUATION 5-4. DATA FLOW EQUATIONS FOR ISOLATED EXPRESSIONS... 39
EQUATION 5-5. FINAL RESULTS FOR OPTIMAL AND REDUNDANT EXPRESSIONS. 39

List of Tables

TABLE 4-1. SCMP GENERAL-PURPOSE REGISTER ASSIGNMENTS. 30
TABLE 8-1. IFFT BASELINE SIMULATION RESULTS. 83
TABLE 8-2. MEDIAN FILTER BASELINE SIMULATION RESULTS. 83
TABLE 8-3. MATRIX MULTIPLY BASELINE SIMULATION RESULTS. 84
TABLE 8-4. IFFT SIMULATION RESULTS INCLUDING PARTIAL REDUNDANCY UNRESTRICTED ALIAS ANALYSIS. 85
TABLE 8-5. MEDIAN FILTER SIMULATION RESULTS INCLUDING PARTIAL REDUNDANCY UNRESTRICTED ALIAS ANALYSIS. 85
TABLE 8-6. MATRIX MULTIPLY SIMULATION RESULTS INCLUDING PARTIAL REDUNDANCY UNRESTRICTED ALIAS ANALYSIS. 86

Chapter 1. Introduction

Compilers have become well established as tools for hardware research. As architectures are developed, the need for application benchmarks soon arises to test new developments. This thesis presents the design and implementation of an optimizing compiler for the single-chip message-passing (SCMP) computer. The thesis begins with a brief overview of the SCMP architecture and the purpose for this research. The chapter also presents the benefits offered by an SCMP compiler.

1.1 Current Processor Limitations

As processor research advances, parallelization becomes of increasing importance. There are two general categories of parallelism: instruction level parallelism (ILP) and thread level parallelism (TLP). ILP attempts to find parallelism among a small set of instructions at execution time, determining how to best reorder them to take advantage of a processor's features. TLP takes a broader approach to parallelism, attempting to break a large problem into smaller segments, usually at compile time, which can be executed in parallel.

Many of today's superscalar processors rely heavily on instruction level parallelism (ILP) to increase performance. Various architectures that attempt to use ILP include out-of-order execution pipelines and simultaneous multithreading (SMT). While some of these attempts have been effective thus far, ILP advancements are now suffering from diminishing returns. As processors grow smaller and architectures become more complex, improvements to ILP techniques become less effective and harder to implement.

Among the reasons for decreasing effectiveness of ILP is the lack of remaining parallelism of instructions. As ILP is generally determined during program execution, the degree of parallelism may be limited by the design of the program. Hartstein, et.al. [Hart02] have shown that most applications perform best with a pipeline depth between twenty and thirty stages. Mitchell, et.al. [Mitch99] have indicated that SMT systems offer little improvement beyond eight simultaneous threads of execution.

The complexity of ILP architectures also makes the task of ascertaining the effectiveness of its improvements difficult. Improvements may have tremendous positive impact on certain applications, almost no impact generally, and negative impact in special cases. Before an implementation of certain ILP algorithms is designed, extensive study must be conducted in an attempt to reveal unforeseen difficulties in the algorithms.

Of increasing concern among superscalar designers is the issue of transistor size and wire latency. Current process technology can use 130nm feature sizes. With these feature sizes, connections between transistors still offer delays at least an order of magnitude

smaller than the switching times of transistors. Within twelve years, a 22nm process technology is predicted, with over 8.8 billion transistors per chip [SIA02]. While these chips supply plenty of transistors to support the increasingly complex ILP architectures, the wire sizes must also shrink, increasing the latencies between transistors. On future chips, designers must then consider the propagation delay of signals over lengthy connections as much as they consider the switching times of transistors.

1.2 An SCMP Overview

The SCMP architecture attempts to avoid the problems of current superscalar uniprocessors by considering parallelism at the thread level [Baker02]. The SCMP processor contains a mesh of up to sixty-four nodes. Each node includes a simple RISC processor, local memory, and a network interface. Communication between the nodes is accomplished through a message-passing network, connecting each node to its neighbors.

To avoid wire latencies, the SCMP processor must consider several design issues that contribute to lengthy connections within a processor:

- Processor Complexity. With superscalar processors, signals may need to travel great distances across the chip. SCMP provides smaller processors that communicate only through the network. No clock or other signals are shared between processors, limiting the length of an interconnection wire to within a small portion of the chip.

- Memory Accesses. Most architectures support several levels of cache and external memory, which signals must often travel off the chip to reach. SCMP incorporates a small amount of memory on each node to hold the program code and data. By keeping the memory close to the processor, SCMP does not suffer from lengthy delays due to memory access.

- Network. The interconnection network avoids lengthy wires by connecting a tile to only its neighboring tiles. Each tile includes a router to deliver messages that must travel greater distances across the chip. Messages are passed through these routers on every clock cycle, limiting the length of network interconnection wires to between neighboring nodes.

There are additional benefits to the SCMP architecture. The RISC processor included on each SCMP tile incorporates a simple in-order pipeline and instruction cache. While not as powerful as the superscalar designs, it is much simpler to implement. The reduced complexity of the processor reduces design and validation time significantly. A new processor may then be replicated across an SCMP chip.

The processor is also easily scalable by incorporating 4, 16, 32, or 64 nodes on a chip. By offering various architectures to serve as the RISC core on each tile, the SCMP processor becomes rather flexible, offering different degrees of ILP within each node.

1.3 Compiler Justification

A major disadvantage to the SCMP design is that it requires at least some degree of thread level parallelism to be effective. How much parallelism is required is difficult to determine. Only by using industry standard benchmarks and specially designed SCMP applications can the effectiveness of the architecture be determined. Compilers offer several advantages in developing these applications.

The obvious benefit of a compiler is in code development. Programming using high-level languages is much more efficient than assembly language programming. There are a host of benchmarks written in the C programming language, and new programs can be developed much more quickly. While general parallel programming skills are required, no knowledge of the SCMP architecture is required to write programs for it. Debugging is also simplified with the use of standard input and output libraries.

A less obvious benefit of a C compiler has to do with changes to the architecture. As SCMP architecture research advances, the need for change to the architecture arises. With hardware changes also come changes to the instruction set. New instructions are added and old instructions are removed. Some valid instructions change their meaning, and still others change in the number of operands required. For assembly code writers, maintenance of old programs proves to be a problem over and above the complexity of writing new assembly routines. As the number of assembly benchmarks increases, the time consumed in updating these benchmarks becomes unmanageable.

With a compiler generating virtually all of the assembly code for SCMP programs, the compiler can accommodate any architecture changes. Modifications to the instruction set, to instruction formats, and to valid data types can be handled rather easily. Once a modification of the compiler is complete, any benchmarks or other programs can then be recompiled; which leads to faster development times, freeing researchers of the details of SCMP programming.

1.4 Thesis Organization

Before we can begin discussing the specific design considerations for our SCMP compiler, we must understand the architecture. Chapter 2 presents a description of the SCMP architecture, along with details we will need to implement a compiler.

A discussion of the compiler development follows. Building an entire compiler from beginning to end is far beyond the scope of this paper. To avoid "reinventing the wheel", we make use of the SUIF compiler infrastructure for which we must only develop a "backend" for the SCMP architecture. Chapter 3 discusses the SUIF architecture and the additional work we must provide to develop our SCMP compiler.

We present the SCMP backend development in Chapter 4.

Chapter 5 and Chapter 7 present two common optimization algorithms: partial redundancy elimination and instruction scheduling. We present these two algorithms because of the locations they occupy in compilation order: before and after register allocation. Combined, they produce an overall reduction of 15-36% in execution time. Chapter 6 discusses some of the difficulties in implementing partial redundancy elimination using the SSA form that are not addressed in previous algorithms. In each discussion of compiler optimizations, we also present necessary changes to SUIF and MachSUIF passes, and discuss requirements of intermediate code. We reserve these discussions until they become necessary, so as to more clearly explain why they are needed.

Chapter 8 presents the test results of all this development, as well as a discussion of the results. Chapter 9 concludes by outlining the work this thesis leaves unfinished and by proposing additions to the SCMP compiler.

Chapter 2. The SCMP architecture.

The SCMP architecture incorporates two very distinct functions into one system. Within each SCMP node lays a simple RISC processor. Sixty-four of these RISC processors are connected via a message-passing network, creating a parallel computer. Independently, these are two research areas that have been dealt with greatly. What distinguishes the SCMP architecture is that these two concepts are combined and examined as one unit.

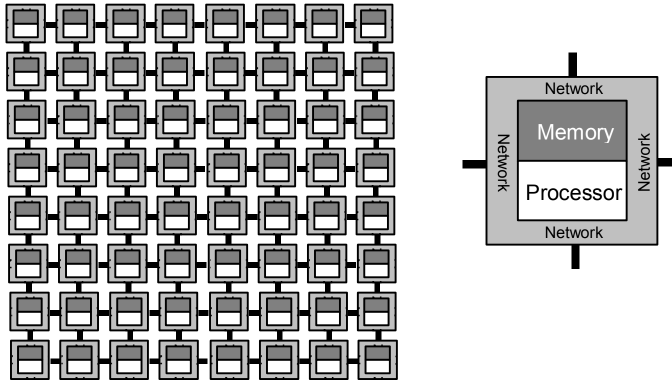


Figure 2-1. A single chip message passing (SCMP) network.

In most parallel computing systems, the processing core is developed without a network in mind. Registers, pipelines, communication, and compilers are developed strictly for general purpose computing. Likewise, most network developers do not give consideration to the machines that transfer data through their networks. The result is a system that works, but with a great deal of overhead incurred to permit the computers use of the network that combines them. Much of the potential for parallel computing is lost.

SCMP seeks to minimize the overhead of a parallel computing system by incorporating the processor and the network, designing each to accommodate the other. While combining these devices has design issues that are prohibitively complex for general-purpose computing, the benefits with certain applications can far exceed the costs.

2.1 The Node

Figure 2-2 shows the general layout of an SCMP node. It contains memory, a basic pipeline and arithmetic-logic unit (ALU), registers, and a network interface. We now examine these components to determine how the SCMP node operates.

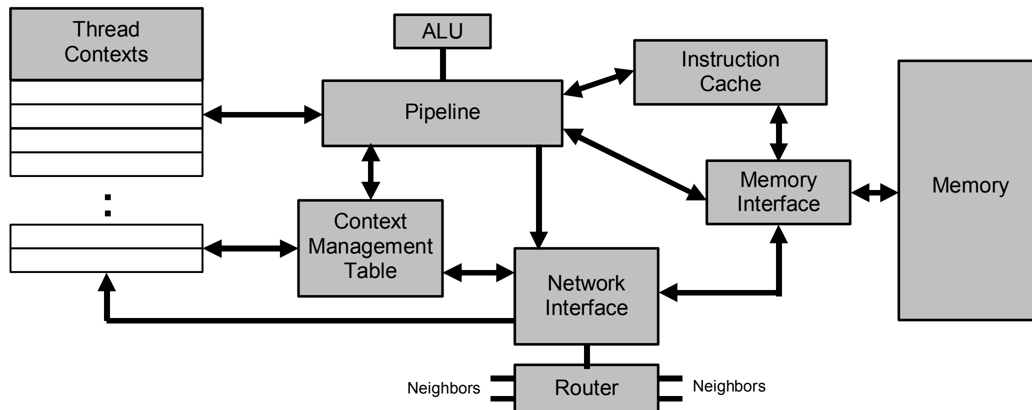


Figure 2-2. Primary components of an SCMP node.

2.1.1 Registers.

The SCMP register set consists of groups of 32 general-purpose integer registers and eight special registers. The special registers contain such information as node identification and network dimensions. While some of these registers can be written, they are reserved for specific purposes and can only be accessed via the special-register SCMP instructions. All other instructions access the general-purpose registers.

2.1.2 Memory.

Each SCMP node contains a small amount of random access memory for program code and data storage. The amount of RAM on each node will be limited to about 8MB. Since all communication between nodes is accomplished through the network, nodes do not share memory. This design has significant implications for SCMP compiler design and programming.

Disadvantages include data and program code replication, excess communication, and limited local memory. Since there is no shared memory between processors, each node must store the program code it is to execute and the data it must process in its own local memory. This repetition consumes a great deal of memory, making memory usage a concern for SCMP application design. The lack of shared memory also mandates that any communication between nodes must be made explicitly. The programmer must include the code necessary for data sharing and synchronizations. SCMP hardware provides no implied service to these ends.

While these advantages are common to all distributed memory multiprocessors, SCMP also includes a limitation of the amount of local memory on each node. Since memory must be on-chip, each node’s memory can occupy only the area around the node, allowing room for network communication. Furthermore, the memory cannot be expanded. Once a chip is made, local memory to each node cannot be added. To expand the memory, one must replace the chip.

Advantages of the local memory include fast memory access. Current predictions indicate that memory can be accessed within two clock cycles. Since these accesses can be pipelined, instructions not dependent on a memory access can be placed after the access instruction, eliminating nearly all memory access latencies. We will discuss this technique, referred to as instruction scheduling, in Chapter 7.

Advantages also include the potential for allowing different nodes to execute different programs. Though current SCMP simulations place the same code on each processor, applications may be able to distribute program code in the same way data is distributed. Each node may provide a specific processing function, eliminating the need for program code replication while freeing local memory.

SCMP memory can be byte (8-bit), half-word (16-bit), and word (32-bit) accessed. The pipeline, the network interface unit, and the instruction cache share access to this memory. A memory controller on the SCMP node manages the accesses.

Memory is one of the most restricting resources of the SCMP architecture. Since each node is limited to its own memory for storage of all program code and data, and since much of this program code and data must be replicated between nodes, memory usage can be surprisingly excessive. Some applications may consume too much memory with program code, limiting the amount of data that can be processed. Other applications may be small, yet operate on large sets of data. In either event, careful data distribution and extra network communication may be required to efficiently make use of the SCMP architecture. Research is currently being done on SCMP application development and how the limited memory affects performance.

2.1.3 Instructions

The SCMP instruction set consists of 76 instructions. The instructions are broken into several categories: arithmetic-logic instructions, control-transfer instructions, memory access instructions, and network interface instructions.

All arithmetic-logic instructions are performed between registers. Therefore, all arithmetic-logic operations are 32-bit operations. Most of these instructions have a variant that allows the operation to be performed with a register source and an immediate value that is stored as part of the instruction. The immediate value is a 13-bit sign extended integer. The result of an arithmetic or logic instruction is always saved to a register.

Control-transfer instructions include test-and-branch instructions, procedure call and return instructions, and all other instructions that can cause the path of execution in a program to change or stop altogether. All test-and-branch instructions compare the value of a register to 0. For example, *`ble r5, branch`* will transfer execution to *`branch`* if *`r5`* is less than or equal to 0. Arithmetic-logic instructions are provided to compare two values, allowing a processor to branch on the result of the test. Our compiler must translate most branches into two instructions: a test and a branch as we will see in Chapter 4.

Memory access instructions are reserved for reading and writing local memory. Address calculations are performed using a base-displacement addressing scheme. The base is some general-purpose register and the displacement is a signed immediate 13-bit integer. The displacement is added to the value of the base register to determine the address for a memory access.

Network interface instructions are provided to send messages from a node. The instructions do not write any registers, nor do they access memory directly. These instructions are permitted only to send register values to the network interface unit as requested. We present more detail on how messages are assembled and delivered to the NIU when we discuss the SCMP network.

2.1.4 Pipeline

SCMP implements a simple in-order execution four-stage pipeline. Assuming no stalls exist, the pipeline enables instructions to be completed on each clock cycle. Certain circumstances may cause the pipeline to stall. Such examples are data dependencies on memory reads, taken branches, and multiple cycle instructions. Some pipeline stalls are potentially avoidable, while others are not [Rama1].

The following are the SCMP instructions that may cause a pipeline stall.

- Memory Accesses. SCMP memory accesses are predicted to take two clock cycles to complete. The memory controller manages these accesses, freeing the pipeline to perform process any following instruction on the second cycle. If the following instruction requires the result of a memory load, the pipeline must wait one cycle for the value to return. Otherwise, a memory access does not stall the processor.

- Multiplies and Divides. The SCMP functional units cannot complete multiples and divides within one cycle. Current predictions estimate that six cycles are needed for multiplication. This delay causes a five-cycle stall in the pipeline. Likewise for division, twenty cycles are needed, causing a nineteen-cycle stall. These stalls are due to limitations within the pipeline, and are independent of following instructions in the pipeline.

- Branches. The pipeline performs an implied branch prediction by the nature of its design. Since the pipeline always attempts to process instructions immediately following a branch, all branches are assumed not taken. Since most branches are in fact taken, the pipeline must stall after each. However, the pipeline stall due to a branch is one clock cycle, so more aggressive branch prediction is likely not worthwhile.

In Chapter 7, we attempt to avoid these pipeline stalls by reordering instructions. We present a more detailed discussion of the pipeline then.

2.1.5 Contexts and Multithreading

The SCMP architecture includes multithreading support by supplying hardware contexts. Each context contains all registers needed to support a thread of execution, including a set of 32 general-purpose registers. A thread has no control over the scheduling of other SCMP threads. It may only initiate a context switch by exiting or suspending itself. The context manager then schedules the next waiting context to become the active context. Scheduling is done in a "round robin" fashion as shown in Figure 2-3.

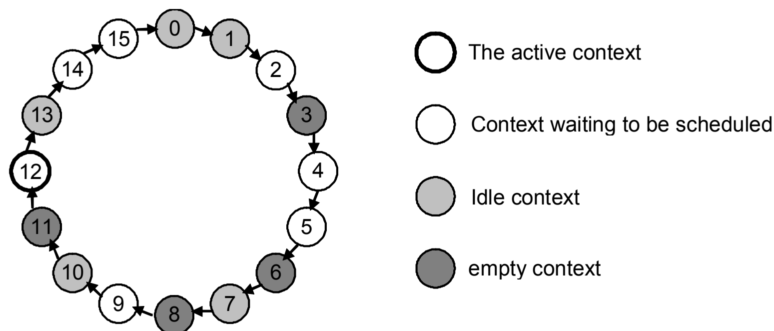


Figure 2-3. The round robin schedule of SCMP contexts. When the thread in context 12 yields the processor, the scheduler will select context 14 as the next context to be run.

SCMP does not support preemptive multithreading. Once a thread begins execution, it continues execution until it exits or suspends. A thread may suspend during network communication, by throwing an exception, or by explicitly executing the suspend instruction. No other means is provided for switching contexts. In the example shown in Figure 2-3, if the thread in context 12 enters an infinite loop, no other threads on the node can execute. Programs must be written so that this cooperative multithreading is managed properly.

2.2 The Network

The SCMP network consists of a four-connected grid of SCMP nodes. Each node is connected with its neighboring nodes to the north, south, east, and west. These connections permit messages to be transferred between nodes. Each node includes a network interface unit (NIU) to control its portion of the network. The NIU allows the node to inject messages into the network, accept messages from the network, and route messages to its neighbors. The SCMP network uses a dimension-order wormhole routing algorithm and virtual channels to help prevent deadlocks in the network and improve its performance [Gold03].

The SCMP network supports two types of messages: data and thread messages. We determine the type of message with the first word of data sent for the message.

- Data messages allow a node to send blocks of data to other nodes and have that data stored directly into the receiving nodes local memory. The sending node specifies the address where the first word should be stored in the receiving nodes

local memory. The sender also supplies an address interval for storing each data word. The receiving node's processor receives no notification when a data message arrives or when it completes.

- Thread messages allow a node to have procedures execute on other nodes. Making a procedure call so that the procedure executes on another processor is referred to as a Remote Procedure Call (RPC). This calling node supplies the address of the procedure to be called on the receiving node. Return information and program arguments are supplied by subsequent sends from the caller.

  When a thread message arrives at the receiving node, a context is allocated and the remote procedure address is stored to the instruction pointer. The following data in the thread message is stored into the context's registers beginning with `r0`. The context remains idle until the message is complete. Once the message is complete, the new thread is scheduled in the normal round robin sequence. In Figure 2-3, if a new thread is assigned to context 6, it will not begin execution until after context 5 is processed.

2.3 External Communication

To be useful in any application, the SCMP processor must support methods for communicating with hardware that is not on an SCMP chip. External control logic, mass storage devices, and data collection tools comprise only a small portion of potential devices that may need to be connected to an SCMP processor. As the type and number of external devices will likely depend upon specific applications, SCMP must provide a general method for communication to maintain its versatility.

One solution for solving this problem uses the edge nodes of an SCMP processor for external communication. Since edge nodes have only three neighbors (corner nodes have two neighbors), each of these nodes has at least one unused network connection. The unused network connection can be used for interfacing directly with special devices, and for communication over a PCI or other local bus.

Another solution allows each node access to external hardware. Current SCMP designs allow a processor to send a message to the NIU, which then sends the message to the node's router. An external interface connection can be added to each node, allowing the processor to send messages to external hardware instead.

In either method described above, SCMP becomes even more versatile by allowing connections to multiple external devices. Each node may serve a specific purpose, connecting a SCMP processor to several unique devices. The only limitation to these communication schemes is the speed of the off-chip interfaces, which can be independent of the speeds of the processor and internal network

No formal work has been conducted on these methods of SCMP communication. Indeed, considerations must soon be made as other portions of the SCMP architecture gain maturity. Limitations and benefits of SCMP at large cannot be understood without

clarifying how an SCMP processor is to be integrated into fully functioning system. However, for the purposes of this thesis and other research currently being conducted on the SCMP architecture, these considerations are not necessary.

We have developed an ad-hoc method for simulating external communication solely for the purpose of other SCMP research. The system is described as part of our simulation methodology in Chapter 8.

Chapter 3. A Compiler Overview

Before introducing the compiler infrastructure, a brief discussion of compilers is warranted. As our compiler is similar in design to the well-known Gnu Compiler Collection (GCC) we will make comparisons between GCC's structure and ours.

The job compilers perform can be effectively broken into two phases. The first phase of any compilation process begins with the front end. The front end takes whatever source file presented to it, written in some high level language, and translates the source into the intermediate language of the compiler. This intermediate language is designed to represent high level programming language concepts such as loops and data structures in a manner that is not specific to any language. We can use the same intermediate language (and the same optimization passes) with any programming language so long as we have a front end to compile the language into our intermediate format. GCC's intermediate representation is called "Trees"[GCC02]. The intermediate language chosen for use by the SCMP compiler is the Stanford University Intermediate Format (SUIF). Any optimizations requiring high level programming concepts must be applied at this phase of compilation.

The second compilation phase begins with a translation from the high level intermediate language to a lower level representation. This lower level language provides a representation that more closely matches the features of the target architecture. Concepts such as loops are transformed into tests and branches, while structures and arrays generally become memory accesses. GCC's lower level representation is referred to as RTL (register transfer language)[GCC02]. In the SCMP compiler, Machine SUIF (MachSUIF) is used. MachSUIF is an extension of the high level SUIF representation. In this phase, general optimizations such as constant propagation and common subexpression elimination are applied. We specify the architecture for which we are compiling, the target architecture, while processing the low-level intermediate representation. We apply target specific code generation, register allocation, and a few additional optimizations after specifying the target architecture. Finally, the source file is output in assembly form. This output file can then be passed through a target specific assembler and linker to produce an executable object file.

3.1 SUIF Introduction

The Stanford University Intermediate Format (SUIF) is an intermediate instruction format designed to represent computer programs at all levels of complexity [Lam1]. SUIF is able to take programs written in high level languages such as C and FORTRAN (from the front end) and transform them into assembly instructions (through the backend) while permitting detailed program transformations at arbitrary intermediate levels.

Originally released in 1994, a second version of SUIF (SUIF2) was released in 1999. SUIF2 is designed to support SUIF1 while adding greater ability to serve as a research compiler. C and FORTRAN front ends have been developed for SUIF2, and with

extended object oriented program support, a JAVA front end has also been released. Another goal of SUIF2 is to be easily extendable into new areas of compiler research. Work has been conducted using SUIF2 with various architectures, including RISC, shared-memory, embedded systems, and programmable logic devices [An1, Babb99].

Though SUIF2 is designed to supplant the original SUIF1, research continues with both releases. For the SCMP compiler, we use SUIF2, as MachSUIF requires it for the backend of our compiler. Any further reference to SUIF in this thesis refers to SUIF2.

Like most other compilers, SUIF is modular. It can be broken into individual steps or passes, each pass transforming intermediate code in some small way. The distinction of SUIF lies in the degree of its modularity.

All passes exist in the form of shared object files that are individually loaded by a SUIF software driver and applied to the program. SUIF passes can be applied multiple times and in any order, as specified by the user. Modifying the order of passes amounts to changing the list of options that is passed to the SUIF driver.

As SUIF contains various loops and structures that cannot be directly translated into machine code, a SUIF intermediate file must be transformed to replace complex high level information produced by the front end with simpler, machine instruction like statements. This "lowered SUIF" transformation actually consists of several passes that must be applied to a SUIF file. An example pass is dismantle_for_statements. This pass converts any for statement in the intermediate code into a series of tests and branches that can later be translated into machine instructions.

```
dismantle_call_expressions
dismantle_field_access_expressions
dismantle_structured_returns
compact_multi_way_branch_statements
dismantle_scope_statements
dismantle_ifs_and_loops
flatten_statement_lists
rename_colliding_symbols
insert_struct_padding
insert_struct_final_padding
```

Figure 3-1. SUIF transformations required to prepare a SUIF intermediate file for MachSUIF.

Figure 3-1 includes the most basic list of SUIF passes that must be applied to a SUIF file to transform it into "lowered SUIF". We will discuss these passes as necessary. Once a SUIF file has been transformed into lowered SUIF, it is ready for further compilation by the backend into assembly code.

3.2 MachSUIF Introduction

Machine SUIF (MachSUIF) is an extension of SUIF that is designed to ease the process of developing a compiler backend for a new architecture [Smith1]. Like GCC's register transfer language, MachSUIF aims at representing machine instructions as opposed to loops and data structures as in SUIF.

Underlying MachSUIF is SUIF, the intermediate language we use for high-level intermediate code. MachSUIF uses SUIF's infrastructure to apply its passes. What MachSUIF adds to SUIF is a simplified interface. Much of SUIF's complexity becomes needlessly cumbersome when dealing with only machine instructions. MachSUIF alleviates this complexity by providing access to a limited set of SUIF's functionality. This makes learning MachSUIF easier, and aids in learning SUIF as well.

Once the lowering passes listed in Figure 3-1 have been applied to a SUIF intermediate file, MachSUIF then translates the intermediate language into the SUIF Virtual Machine (SUIFvm) [Holl1]. SUIFvm represents a virtual RISC architecture. Its advantages as a virtual machine are that it has an infinite register set and it avoids the minute details required in implementing a compiler backend. We perform all general low-level optimizations on SUIFvm intermediate files. Examples of these are common expression elimination, constant propagation, and dead code elimination. As we use SUIFvm instructions in many of our examples, we provide a portion of the set in Appendix A.

A translation of SUIFvm code to the target architecture follows. For our compiler, the target is the SCMP architecture. Register allocation follows this translation. After register allocation, we layout the stack frame and produce assembly code.

Figure 3-2 includes a list of that generate an assembly file from a lowered SUIF file. Passes that require information specific to the target architecture are indicated.

`s2m`	Translation of lowered SUIF to SUIFvm intermediate code
`scc`	Constant Propagation
`dce`	Dead code elimination
`gen -target_lib <target>`	Translation of SUIFvm to target specific instructions
`raga`	register allocation
`fin`	stack frame layout
`m2a`	assembly code generation

Figure 3-2. MachSUIF transformations required to produce assembly code

The assembly code produced by the compiler must then be assembled and linked with other libraries to produce the executable file. The work of our compiler concludes with the passes in Figure 3-2.

3.3 SUIF and GCC

GCC and SUIF have been considered to serve as the main infrastructure for the SCMP compiler. As they have many similarities in their designs, some justification for the decision to continue work with SUIF is warranted.

As mentioned earlier, SUIF and GCC share a high level organization with two basic compilation phases, and they both are supported under a wide range of Unix based operating systems. However, each infrastructure has its advantages and disadvantages.

3.3.1 GCC's Advantages.

Stability.

- A SUIF user will find himself immediately among the SUIF beta testers and debuggers. Although serious problems in the structure of SUIF are rare, small problems surface sufficiently often to slow development. The same difficulties apply to MachSUIF users. Support continues, and problems are often resolved quickly. However, the possibility of errors within SUIF make for difficult debugging of new compiler passes.

- While GCC is under constant development, and latest versions may not be reliable, GCC still provides a very stable compiler system. There are a great number of developers who do a very good job at finding and solving obscure problems. This is essential, as compiling programs proves to be a very large and complex task.

Language Support.

- SUIF supports C directly with indirect support for FORTRAN77 and JAVA. SUIF's C front-end is provided as closed source object code. Problems with the SUIF frontend cannot be corrected, and the frontend cannot be modified for the development of new SUIF front-ends.

- GCC readily supports C, FORTAN, and JAVA, C++, Objective C, and ADA [GCC02]. GCC's frontends are enitrely open source, allowing programmers to use the source to develop new languages or extend old ones.

Optimizations.

- The SUIF distribution includes very few optimizations. When combined with MachSUIF it provides simple optimizations for function inlining, dead code elimination, constant propagation, and register allocation. Other SUIF developers have written passes, but the passes are rarely made available to the general public, and are not reliable.

- GCC provides a host of robust optimizations, including stronger versions of function inlining, dead code elimination, constant propagation, and register allocation. GCC also provides two styles of expression elimination, instruction scheduling, and basic block scheduling [GCC02].

3.3.2 SUIF's Advantages.

Object Oriented Design.

- GCC is written almost entirely in C. Though this makes them very fast, programs written in C can be very difficult to understand. A programmer with little knowledge of compilers will find the structure of GCC source code very difficult to modify.

- SUIF is written in C++ with a thorough class structure. While not necessary, it does aid in understanding the structure of SUIF and of compilers in general. With a good, clean design, SUIF allows programmers to begin modifications while knowing less about the overall structure of compilers. In this respect, SUIF does a good job of "teaching" programmers how to write compilers.

Parallelization

- GCC has been designed as a general-purpose compiler. It is not designed as a research compiler or as a parallelizing compiler, both of which are important in performing SCMP research. While GCC provides effective sequential optimization techniques, it is not accompanied by research in other compiler areas.

- SUIF is designed for automatic parallelization of sequential program through data and control flow analysis, program partioning, and other tools [Laio99, Ras98]. As our eventual goal in developing a compiler is to improve parallelization techniques for the SCMP architecture, the tools SUIF offers can be of great asset.

Modularity.

- As SUIF is designed as a research compiler, it is designed with constant changes in mind. Together with its object-oriented design, its modularity makes it much more quickly understandable. Its modularity also allows recompilation of the compiler due to small changes within it.

3.3.3 The Final Decision

Ideally, a compiler using GCC and another compiler using SUIF would be developed to compare their performance. However, this paper focuses on the design and implementation of one functioning compiler. As SUIF provides more flexibility and easier redesign, it lends itself more to a research environment. SCMP compiler research continues using the SUIF infrastructure.

3.4 The Need for Optimizers

With no optimization, compilers tend to produce very poor code. Evaluation of constant expressions will be done at runtime. Multiple accesses to an array element will translate into multiple calculations of the element's memory address. A processor may provide advantages that a compiler cannot realize without special considerations. None of these issues are addressed in the *required* elements of a compiler. It must work only well enough to generate correct code.

For this reason, optimizations have become an established part of all compilers. In our SCMP compiler, we design and implement two common and effective optimization passes: partial redundancy elimination (Chapter 5) and instruction scheduling (Chapter 7). A sparse conditional constant propagation (SCC) pass and dead-code elimination pass are included in all baseline tests [Weg91, Morgan98]. Also included is register allocation and decode elimination as supplied with the MachSUIF distribution. Documentation can be found on MachSUIF optimization passes in the MachSUIF documentation [Smith1].

These algorithms are well known and well established in many current compilers. We find, however, that there are certain problems with these algorithms that are either omitted or are not encountered in other discussions. Some of the problems are attributed to the uniqueness of the SCMP architecture, while others can be found in any multiprocessing environment. We attempt to present these issues with our implementations of partial redundancy elimination and instruction scheduling, along with our solutions.

3.5 Compiler Summary

SUIF and MachSUIF serve as an extensive infrastructure for designing compilers. SUIF provides the necessary components for beginning the compilation process, interpreting handwritten C code into an intermediate language. MachSUIF provides a simplified intermediate language for developing the compiler's backend, where our SCMP specific developments are made.

Though other compiler infrastructures can supply the same benefits as the combination of SUIF and MachSUIF, these tools provide the strongest advantages in a research environment. SUIF and MachSUIF allow a high degree of flexibility in representation, allowing for unique approaches for compiler development with new architectures such as SCMP. In addition, MachSUIF provides the tools we need for performing low-level optimizations.

Chapter 4. The SCMP backend.

The compiler discussion presented in Chapter 3 deals with the compiler at large, discussing the inner workings of SUIF and MachSUIF. To use this compiler to generate SCMP programs, we must develop an SCMP backend. This backend must translate SUIFvm intermediate code into SCMP equivalent instructions. Fortunately, included in the MachSUIF distribution is an Alpha backend after which we model our compiler [Smith1].

4.1 MachSUIF backend passes.

Recall the list of MachSUIF compiler passes shown in Figure 3-2. They are shown below for convenience with the SCMP architecture shown as the target. Passes involving the compiler's backend are indicated. These are the passes about which we must be concerned when developing our SCMP backend. While the structure of MachSUIF allows us to specify the SCMP architecture without having to worry about each individual pass, it is worth noting what happens during each pass.

```
s2m
scc
dce
gen -target_lib scmp   SUIFvm to SCMP conversion
raga        SCMP register allocation
fin         SCMP stack frame layout
m2a         SCMP assembly code generation
```

Figure 4-1. MachSUIF compiler passes. Passes involving the SCMP backed are shown in **bold**.

4.1.1 Code generation

The code generation pass (gen) is the first pass in our compiler in which the SCMP architecture is considered. All MachSUIF passes prior to the gen pass operate on SUIFvm instructions. During code generation, we convert SUIFvm instructions into SCMP instructions. Instruction operands remain largely untouched, so virtual registers and variables still exist after code generation.

Some of the SUIFvm instructions can be translated directly into SCMP instructions. Translating the SUIFvm instructions in Figure 4-2 amount to merely replacing each with the proper SCMP opcode.

```
add s5, s6, s7         add   s5, s6, s7
sub u3, u4, 10         subui u3, u4, 10
mul s1, s2, -12        muli  s1, s2, -12
sle s7, s8, s9         sle   s7, s8, s9
```

Figure 4-2. SUIFvm instructions and their SCMP equivalents

Since all SCMP branches compare one register to zero, SUIFvm branches must be translated into at least two SCMP instructions: a test and a branch. Memory to memory transfers require a load and a store in the SCMP architecture. Figure 4-3 shows examples of these translations.

```
blt r7, r8, label        slt r6, r7, r8
                         bne r6, label

memcpy r1, r2            ldw r3, 0(r1)
                         stw 0(r2), r3
```

Figure 4-3. SUIFvm instructions requiring multiple SCMP instructions

Most SCMP instructions can be translated from SUIFvm instructions during code generation. Some SUIFvm instructions require special consideration in the SCMP architecture. In section 4.6, we discuss these instructions and our approach to handling them within the compiler.

4.1.2 Register Allocation

The register allocator uses the information given concerning the available registers and the SCMP calling convention to determine how to most effectively use the SCMP register set. Of course most programs will require more registers than the SCMP offers. For this reason, the register allocator must often insert register spills (storing register values to memory) to save values for use later in a procedure. Values are stored on the stack. The register allocator does not determine the position of each spill on the stack. It only indicates where spills are necessary, leaving the rest to further compilation.

4.1.3 Code Finalization

The job of code finalization (fin) is to layout the stack frame of each procedure call. During the fin pass, locations are assigned on the stack for each value that must be spilled from registers, requiring that code finalization be done after register allocation. The fin pass completes setup of our calling convention. This completes the compilation process. No significant transformations occur to the code once the fin pass is completed.

4.1.4 Assembly Code Printing

The final step in our compiler involves printing the resulting assembly code. MachSUIF allows us to generate additional instructions during assembly code printing. However, adding instructions while printing makes debugging more difficult, as the same process is used to print intermediate results. Therefore, we avoid having our SCMP printer perform any code transformations.

4.2 Calling Convention and Register Usage

We begin the discussion of the SCMP backend by describing the calling convention. An architecture's calling convention describes how the processor manages registers and stack

space between procedure calls. The calling convention establishes locations for return address, return values, saved registers, and local variables. While describing our calling convention, we use two common terms: the caller and the callee. In a procedure call, the caller makes the procedure call, while the callee is the procedure being called.

We begin the discussion with the calling convention as it determines the behavior of our compiler at all stages after the SCMP target has been specified. The gen pass must know how many registers are reserved for procedure arguments and which register is reserved for the stack pointer. The register allocator must know whether registers are caller or callee saved. The fin pass must then take all information given by previous passes and determine where values should be stored on the stack. We will get to the details of all these problems shortly, but it is important to point out that the calling convention determines a great deal about how the SCMP backend should be written.

In a RISC architecture, the calling convention must define how registers are used. Before calling a procedure, the caller must know which register values will be valid upon the procedure's return. The callee must know where arguments are stored. The callee must also know how to return to the caller, and how to return a value if necessary.

When discussing register usage, we will use two more common compiler terms: caller-saved and callee-saved registers.

- A caller saved register is one in which the caller must preserve the register’s value by some means (usually by storing it on the stack) if it wishes to retrieve the register's value after a procedure call. The callee assumes that the caller preserves any caller-saved registers, and that it can modify these registers without preserving them again.

- A callee saved register is one in which the callee preserves the register's value before modifying it. The value is then restored to the register before the procedure returns. The caller can assume that callee-saved register values are the same before and after a procedure call.

The SCMP register usage is modeled after that of the Alpha processor [ALPHA]. The register set is divided into three classes: argument registers, temporary registers, and saved registers. These categories are defined as follows:

Argument registers. If all arguments to a procedure must be stored on the stack, the caller must store the arguments and the callee must load them. Allowing register storage for procedure arguments avoids some, if not all of these stores and loads. Avoid stack storage also reduces the size of the stack. Since the caller does not often reuse arguments to a procedure, argument registers qualify as caller saved registers.

Temporary registers. Temporary registers serve as temporary placeholders for computations. Since the values they store are considered temporary, a procedure needs not save a temporary register before modifying its contents. This makes temporary registers caller saved registers.

Saved registers. Saved registers are intended to store values that a caller may need after a procedure call. The caller does not need to worry about the contents of saved registers, as these registers must contain the same value upon exit as they do upon entry. Saved registers are of the callee saved class.

Having both caller-saved and callee-saved registers presents a balance between the advantages and disadvantages of either register class. In a general purpose computing environment, a caller has no knowledge of the callee's register usage, nor does the callee have any knowledge of the caller's register usage. If all registers are caller-saved, a caller must save all registers containing values that are needed after a procedure call, even if the caller uses no registers. If all registers are callee-saved, a callee must save any registers before it uses them, even if the caller does not need any to have any values preserved. Having some combination of both classes allows us some discretion during register allocation as to what type of register to use.

The SCMP register set is divided into the classes described above. However, not all of the 32 general-purpose registers can be used as argument, temporary, and saved registers. We must reserve several registers for special purposes such as procedure return values, return addresses, and stack pointers. Here we discuss these reserved registers and why they are needed.

Stack Pointer. Like many RISC architectures, the SCMP processor does not have a dedicated stack pointer. However, given that most programming languages make use of a stack to store local variables and any excess arguments, we must designate a register to serve as the stack pointer. Note that this register must be initialized to a block of reserved memory. SCMP stack organization and initialization methods are discussed in sections 4.3 and 4.2.

The stack pointer is used frequently. Local variables, parameters, and register storage is located on the stack, and stack positions are referenced from the stack pointer. Therefore, the stack pointer must always be the stack pointer, preventing it from being used among any of the register classes.

Return Address. Whenever a procedure is called, an address must be stored to indicate where execution should resume after completion of the procedure. With SCMP procedure calls, we reserve a register for this purpose. The register is specified within an SCMP procedure call instruction. In the example below, the return address is stored in r5.

```
bsr r5, _procedure_
```

Since the return address can be stored in any register, it is possible that we do not have to designate a specific register for this purpose. However, to return from a procedure, the return statement must be supplied with the correct return address. By reserving a single register for this purpose, we can supply the same return address register for all procedure calls and returns.

The return address is only needed at the conclusion of a procedure. It may seem profitable to classify the return address register as a caller or callee-saved register and use it otherwise during the procedure. However, we find problems exist with either approach.

- When using the return address register as a callee-saved register, the callee will store the return address before modifying the register. When a procedure call is made, the return address register is assigned with a new return address before the callee saves the register. If the register allocator cannot determine that the register is assigned during the procedure call, it will assume that the callee maintains the register's value. Upon return, the register contains the previous call's return address, and the any value held by the register before the call is lost.

- When using the return address register as a caller-saved register, the caller will preserve the register's value before making the procedure call. However, the callee will not preserve the register's value before making its first assignment to the register, thereby destroying the procedure return address. We can require that callees always preserve their return address register, but then we must require that all procedures have a stack. This imposes a greater restriction on the compiler than does reserving the return address register.

For these reasons, we reserve the return address register for exclusive use. The register allocator will have no freedom in using this register for other purposes. This pass is labeled as `dismantle_structured_returns` in Figure 3-1.

RPC Return Address

A remote procedure call (section 4.3) requires two components for a return address: the return node id and a callback address. We can use the return address register to store the callback address. However, storage for the return node id requires an additional reserved register.

Only upon remote procedure return is the return node id needed. It can be modified freely otherwise. While it is possible to use this register as a caller saved register, it is simpler to make it callee saved. As a callee-saved register, a remote procedure automatically saves the register's value and restores it before the procedure completes. Processing for remote procedure initialization and return occurs outside the main procedure's body, making the register's general-purpose use transparent.

Return Value. We also must reserve a register for returning a procedure value. Any procedure that returns a value must then store that value to this register before returning. An example declaration of such a procedure is

```
int returns_a_val()
```

Since this register is only needed at the end of a procedure, if at all, the register can be used as a general-purpose register during program execution. Only at the conclusion of a procedure is the register used to return a value. Since the register will likely be written

and by definition cannot be restored by the callee (else a different value could not be returned), the register is used as a caller saved register. The caller preserves it when necessary allowing the callee to use it freely.

Special exceptions must be made for returning values that cannot fit within the return value register. An example is a procedure that returns a structure. To handle this problem, we convert any procedure that returns a value larger than 32-bits into a procedure that takes as its first argument a pointer to a variable local to the caller. Upon return, the callee copies all results to the compiler generated structure and returns void. An example of the transformed procedure is shown below.

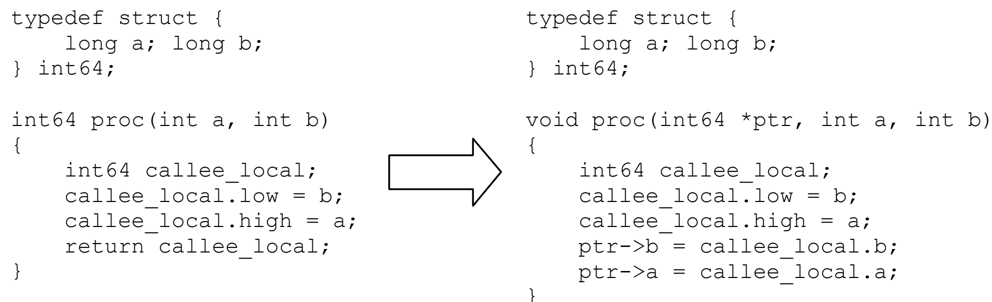


Figure 4-4. Effective transformation of a procedure returning a structure

This transformation requires a high level representation of the procedure in question. Therefore, we implement the transformation as a SUIF pass.

4.3 Remote Procedure Calls.

Most programming languages do not provide the ability to call procedures remotely as part of the language. Libraries must be built around the processor architecture and network. With this approach, remote procedure calls are generally an after thought. Handling these special calls simply means developing wrappers around the communication support that the network offers.

With the SCMP architecture, remote procedure calls (RPCs) are built in to the processor ISA and the network. Not only are RPC libraries not ideal for this kind of architecture, they are insufficient for supporting the advantages SCMP has to offer. For this compiler, we must develop a method for embedding RPCs into the programming language.

Designing a system to support remote procedure calls presents two majors issues that must be addressed:

- The caller's problem: How to correctly call a remote procedure and receive notification that the procedure has completed.

- The callee's problem: How a remote procedure should be written so that it can be called remotely and so that it can send notification of completion.

We deal with the callee's problem first, as its solution is a bit more concrete. The caller's problem has no good definition, and so a solution is almost implementation dependent.

We must examine the callee's problem in a bit more detail. The problem arises as most work in handling the RPC is embedded into the SCMP architecture. The compiler has little control over how the call is made, so it must try to insert code to solve any problems. To understand these problems, a discussion about SCMP remote procedure calls is warranted.

```
sendh r6, THREAD, r7
send2 r5, r11
send2e r12, r10
```

Figure 4-5. SCMP instructions to generate a remote procedure call.

Here we have sample assembly code for making a remote procedure call with SCMP instructions. r6 contains the destination node and r7 contains the address of the function to be called. r5, r11, r12, and r10 contain 4 parameters to be passed to the function. A message is assembled and delivered by the sender. Upon receipt of the message, the destination node follows these steps to begin the procedure:

1) Allocate a new context and assign it to the new thread.
2) Store the procedure's address in the instruction pointer.
3) Store the procedure's parameters in registers `r0`, `r1`, `r2`, and `r3`.
4) Begin execution.

All of these steps are performed by hardware. During this initialization, the node continues processing the active thread. It will not begin executing the new thread until it is scheduled according to the SCMP's scheduling algorithm (section 2.2).

If standard C procedures are called remotely in this manner, here are several issues that cause major problems.

- No stack has been allocated. Since the stack is used to hold local variables and to call other functions, a remote function will modify the stack pointer to allot space for itself. However, the new thread has not initialized the stack pointer. The procedure will use whatever address the stack pointer contains as the thread's stack, which will probably cause the program to fail.

- When a procedure ends, it will attempt to return as if it has been called as a standard procedure. RPC conventions dictate that an RPC must call a remote procedure on the node initiating the RPC, if requested, and then terminate it's thread. The calling node id and callback function address are supplied as two arguments in the RPC. Returning to the callback procedure as if it is local will likely prevent correct synchronization.

4.3.1 Stack Allocation

When a thread begins, its stack pointer is not initialized. The register may point to an invalid address, another thread's stack, or some other important data. As the thread begins execution, it will likely begin modifying the stack immediately, regardless of the validity of the stack pointer. A remote procedure must be able to initialize the stack before any further processing is done. To do this it must have a block of memory reserved for its stack so that it can initialize its stack pointer.

Therefore, we reserve a block of memory in each SCMP node to serve as the stack space for all the contexts. Since the stack space is contiguous, a program executing in a given context can determine the location of its stack by computing an offset from the beginning of the stack space. By letting STACK_SIZE be the size of an individual context's stack, CTX_ID be the id of the context, and STACK_START be the start of the node's stack space, a context can compute the start of its stack address as follows:

SP = STACK_SIZE * (CTX_ID+1) + STACK_START

Notice that according to this calculation, the stack pointer will actually be initialized to the bottom of the next context's stack. Since usage of the stack begins at the highest address of the stack's memory and grows downward, we initialize the stack pointer so that it begins at the top of the stack.

Figure 4-6 shows the necessary code to be inserted at the head of a remote procedure and an example of how this calculation allows a remote procedure to determine its stack location.

```
readsr tmp1, ATR
addui tmp1, tmp1, 1
llo sp, STACK_SIZE
mul tmp1, tmp1, STACK_SIZE
llo/lhi sp, STACK_START
add sp, sp, tmp1
```

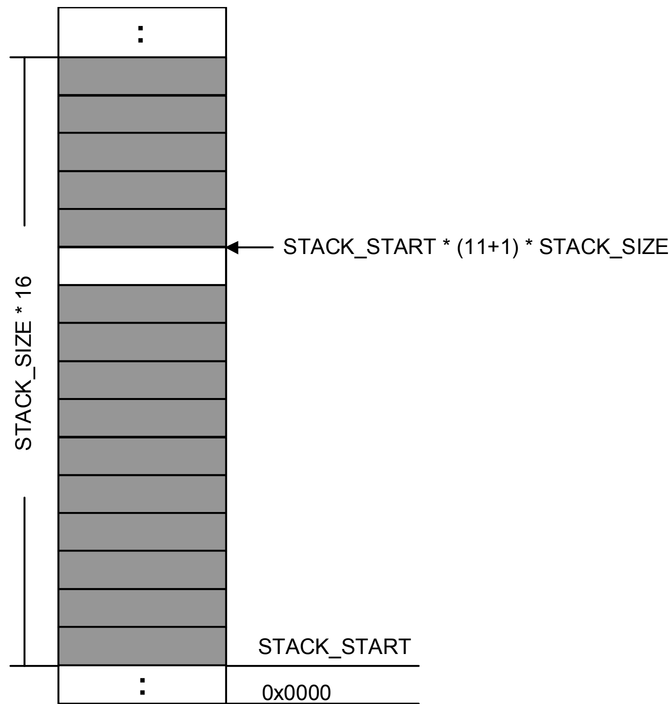


Figure 4-6. SCMP stack organization, with the calculations for locating the stack segment for context eleven.

Certainly, we are not limited to having all threads with the same stack size. Many programs are written such that only one thread uses a large amount of stack space, while many threads perform small computations and require virtually no stack space at all. While it might be more efficient to reserve a large stack space for some context on each node and reserve smaller space for other contexts, this implementation becomes rather difficult.

The disadvantages to our stack allocation scheme are strong. Since there is no concept of virtual memory or other memory management as done by an operating system, the stack is limited to whatever space that is allocated to it initially. Additionally, the memory allocated to the stack will be reserved exclusively for the stack. If the memory allocated to the stack is too small, the program will overrun its stack and crash. If the memory allocated is too large, sufficient memory may be unavailable for other purposes (dynamic memory allocation, data storage, buffering).

4.3.2 Remote Procedure Returns

When a remote procedure finishes, it must notify its caller of its completion and stop execution of the thread. Neither of these actions is required of a local procedure. For correct operation, the SCMP compiler must be able to include this functionality in each remote procedure.

SCMP allows for callback functions as a means for determining completion of a remote procedure call. The caller may indicate its desire for notification by supplying a callback function as part of the thread message. When the remote procedure completes execution, it indicates to its caller this completion by sending a thread message to the caller to execute a callback function. The caller may indicate that it does not desire callback notification by supplying no callback function.

For this callback ability to be supported, the SCMP compiler must include instructions at the conclusion of a remote procedure to check the callback function register (r1) and send a thread message to the caller if desired. If the return address register is NULL, then the caller requests no callback, so execution may end immediately. The instruction sequence for accomplishing this is shown in Figure 4-7.

```
    beq ra, label1
    llo tmp1, 0
    sendh rn, THREAD, ra
    send2 tmp1, tmp1
    sende rv
label1:
    end
```

Figure 4-7. Instructions for calling a SCMP callback function

To ensure proper deallocation of the context, we replace the return statement (RET) with the thread end statement (END) at the conclusion of the remote procedure. Since all execution of the thread stops at this instruction, no subsequent return statement is needed. Note that no stack maintenance is needed on the conclusion of a remote procedure. The remote procedure is complete, and the data stored on the stack is no longer need. Since the static space is allocated statically, it will remain unused until another thread is assigned the context.

4.3.3 RPC Identification

All this discussion of how to handle remote procedures is moot unless we have a way of identifying remote procedures. Fortunately, C provides the pragma directive for including implementation specific information in a C program. By preceding a function name with the line `#pragma remote`, the compiler can recognize the function as a remote procedure and compile it accordingly. Figure 4-8 shows an example of a SCMP remote procedure.

```
#pragma remote
void remote_function( .. parameters .. ){
    ... function body ...
}
```

Figure 4-8. An SCMP remote procedure

4.3.4 Calling a Remote Procedure

We have shown a program someone might write for the SCMP architecture below. There are many ways to make this procedure call. The programmer may wish to call the procedure on a single node, he may wish to call the procedure on all nodes, or he may wish to call it on some set of nodes. The set may include all nodes in a row or column, or it may be a random set of nodes distributed throughout the processor. In any event, we must allow the programmer to call procedures in whatever fashion the architecture allows.

```
#pragma REMOTE
int remote_proc(int xdim, int ydim){
    return process_local_data( xdim, ydim );
}

main(){
    remote_proc( 3, 5 );
}
```

Figure 4-9. Defining and calling a remote procedure

We find that most of the applications written for SCMP follow a common programming model:

> The main routine executes a series of remote procedures on all nodes, allowing each to perform some computations on a certain portion of the data set. Main must generally wait for all nodes to complete a certain procedure before it can request that all nodes execute the next procedure.
>
> Meanwhile, each node makes individual remote procedure calls to exchange data before and after processing. Threads may wait for RPC completion, or they may continue processing during the calls. Each thread must always wait until all RPCs and other messages complete before it can send notification of completion to the main thread.

This programming model resolves some of our RPC issues. We must still be able to support all forms of remote procedure calls, but our main focus deals with solving the problems of calling a remote procedure on one node or on all nodes. We provide a general solution to these problems through several library routines as described in Appendix B. This library allows remote procedure calls from C routines, maintaining reasonable performance while providing the synchronizations necessary to insure correct operation. Of course a programmer may choose to perform all synchronization manually by using the single-node RPC routines.

4.3.5 Language Compatibility

We focused on adapting the C programming language in this discussion, but the same basic problems must be solved for any language. Hopefully, by solving the problems using SUIF and MachSUIF, we have largely solved them for any programming language. However, to take full advantage of the features of SCMP a programming language must be capable of these features.

- the ability to attach such annotations as C's pragma.
- the ability to include inline assembly code or call C functions.

4.4 Register Assignments

With the register classes and reserved registers declared, we must assign each SCMP general-purpose register. Since we must call two different types of procedures; local procedures and remote procedures, we must make register assignments so that the two types of procedures are compatible, as a remote procedure must be able to call a local procedure directly. As we assign registers, we consider local and remote procedures and make register assignments to follow the calling conventions of both.

As discussed in section 4.3.2, each procedure call requires at least two arguments: the calling node's id and a callback function. Even in the event a callback is not requested, NULL values must be sent as these two parameters, as is shown in Figure 4-4. Therefore, we assign registers r0 and r1 as the return node and return address registers. These assignments require that we always send the return node's id and callback function address, in that order, as the first two parameters of a remote procedure call.

Note that this does not imply that a remote procedure must have as its first two arguments the calling node's id and requested callback function. At present, we do not provide a method for allowing a programmer to initialize these values directly from source code. The SCMP routines in Appendix B take as parameters the remote procedure address, callback address, and any additional arguments to be passed to the remote procedure and assembles a thread message so that are stored in the proper registers of the callee.

Argument registers are also assigned according to RPC behavior. When a remote procedure begins execution, it expects its first arguments to be in the argument registers. However, during thread message receipt, all arguments are stored sequentially in registers beginning with r0, regardless of their purpose. This places the first procedure argument, which follows the return node id and callback address, in r2. By assigning r2 through r9 as arguments registers, we avoid having to copy arguments into the correct registers before beginning execution.

When an RPC begins, some non-argument registers may contain the procedure's arguments. Since our special considerations for a remote procedure are only upon its entry and exit, these arguments must be stored on the stack just as they are during a local procedure call.

Of course, this implies that the stack pointer must be initialized. As shown in Figure 4-6, this initialization requires two registers: the stack pointer, and a temporary register. Unfortunately, we cannot preserve the values of these two registers. We then limit the number of arguments than can be sent in a thread message to thirty, using register r31 as the stack pointer and r30 as the temporary register. A thread message may still include up to thirty-two arguments, but the last two are lost during stack initialization.

The locations of save, temporary, and return registers are not as significant. They are assigned the remaining registers as shown in Table 4-1.

register	assignment	classification
r0	return node id	caller-saved
r1	return address	reserved
r2 - r9	argument registers	caller-saved
r10 - r20	save registers	callee-saved
r21 - r29	temporary registers	caller-saved
r30	return value	caller-saved
r31	stack pointer	reserved

Table 4-1. SCMP general-purpose register assignments.

The sizes of our register classes are based upon general register usage. Caller-saved registers are used more heavily by our register allocator than are callee-saved registers, making the availability of caller-saved registers more important. We rarely find that a procedure requires more than eight arguments. Variable argument procedures and procedures passing large data structures are the typical causes of a procedure's large argument size. For virtually all purposes, eight argument registers is sufficient.

4.5 Stack Frame Layout.

Every function needs a stack to correctly operate. For general-purpose computers, a stack is generally allocated for a new thread by an operating system. By the time a function begins execution, it has already been provided an environment so that it can operate correctly.

In the SCMP architecture, there is no concept of an operating system yet. A thread of execution must create and maintain its own stack. The compiler must insert instructions in a program to setup and maintain the stack. For this reason, stack maintenance becomes a problem for which the SCMP compiler must be concerned.

For minimal stack maintenance, we need a stack that can be initialized at startup and used without further care. Simply put, some block of memory must be allocated to the stack at startup, with no further maintenance of the stack after initialization. This lack of concern carries two advantages. The first is simplicity. Startup code must initialize the stack pointer, but no additional compiler design is needed. The second advantage lies in program overhead, or the lack thereof. Less code inserted by the compiler means fewer instructions to be executed and faster execution time.

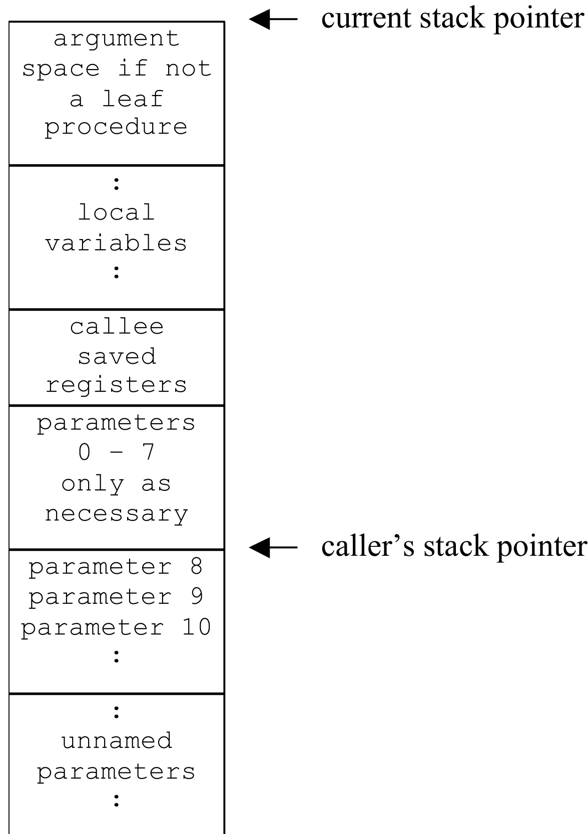


Figure 4-10. Stack organization for an SCMP procedure call.

Parameters are stored in contiguous addresses to ease processing in variable argument procedures. A varargs procedure calculates the offsets of unnamed parameters based on the address of the last named parameter. In the event a procedure has only one named parameter, all subsequent parameters are in contiguous memory locations, making their locations easy to determine.

In the event a varargs procedure has less than eight named parameters, the last named parameter and any unnamed parameters passed in argument registers must be stored to the stack such that all arguments following the last named argument are in contiguous memory locations.

4.6 Macro Instructions

The SCMP backend requires the addition of several macro instructions. These macro instructions are not actually part of the SCMP instruction set as implemented in hardware. The purpose of these instructions is to carry important data flow information through the code generation stage of the compiler. The instructions can then be expanded as necessary before assembly.

Many architectures offer these macro instructions. The instructions exist for a variety of purposes ranging from programmer convenience to data flow analysis. They are

translated into a series of valid assembly language instructions by the compiler or assembler. Many of Alpha's load and store operations are expanded into a series of smaller load and load immediate instructions. These macro instructions can be expanded at almost any point during compilation once an architecture has been defined. Generally, macro instructions must be expanded before a program can be passed through a linker.

When macro instructions are expanded some data and control flow information is gained, while some is lost. Where an instruction should be expanded depends on how the information in the original instruction is needed, and when information for the expanded instruction list is best used.

For example, the Alpha's `lda` instruction may be expanded as shown.

```
lda  $sp, 400,000($sp)    An unexpanded lda instruction.
---------------------
ldah $at, 6               A possible expansion of lda.
lda  $at, 6784($at)
addq $sp, $at, $at
lda  $sp, 0($at)
```

This `lda` instruction attempts to load the effective address of a variable offset 400,000 bytes from the stack pointer. While this information is readily available by observing the original code, determining the intent of the expanded code is not so obvious. In early optimization, a high level understanding of instructions is required. However, an instruction scheduler will find the expanded set of instructions more helpful, as it can arrange them to fill potential stalls in the pipeline.

MachSUIF requires the architecture to be specified during the code generation (`code_gen`) stage. Therefore, macro expansion can occur during code generation and at any further stage in the MachSUIF backend (Figure 4-1). We have several SCMP macro instructions, each with its own ideal point for expansion.

4.6.1 MOV.

Many architectures include a move instruction. It is intended to move a value from one register to another. Alpha, MIPS, and many embedded processors include a move instruction. Even simpler embedded processors usually have some form of move instructions.

For the SCMP instruction set, we might assume that a move instruction is unnecessary, as it can be implemented using an add to an immediate value of 0. A sample translation is shown.

```
mov r5, r6  ⟹  addui r5, r6, 0
```

Our need for a move instruction is due to MachSUIFs behavior. Certain compiler passes attempt to insert register-to-register move instructions during optimization. Without a

valid move instruction, NULL instructions are inserted instead, and the program becomes invalid.

For example, static single assignment may create move instructions when converting a graph from SSA form to CFG form (Chapter 6). The SSA libraries simply request the opcode for a move instruction from the architecture definitions, and a destination and source are appended to the new instruction.

A better approach to this insertion of move instructions might be to create a SUIFvm mov instruction and have the code generator translate the SUIFvm instruction. That would be a more general solution to the problem. At issue is that it requires a redesign of the the MachSUIF code generator. Having a mov macroinstruction proves necessary and sufficient.

Since move instructions may be created at almost any point in the compilation process, and since the instruction expands to a single add with a constant 0, mov instructions may be expanded at any point in compilation.

4.6.2 SET, LDC.

`set` and `ldc` are macro instructions for loading constants. They are supported for backward compatibility. Loading 32-bit constants requires two instructions, `LLO` and `LHI`. The `set` macro was defined in earlier assembly langauge programming to simplify the loading of constants by generating the `llo/lhi` pair when necessary. `ldc` was later added to perform the same function.

Of course the compiler does not need these instructions. Code generation can easily translate from the SUIFvm `ldc` to an SCMP `llo/lhi` pair when necessary. These instructions are offered to programmers wishing to use them in inline assembly code.

As any constant propagation is done with SUIFvm intermediate code, we find no advantage to supporting `set` or `ldc` beyond the code generation stage. These macro instructions are expanded during code_gen, just as the SUIFvm `ldc` instructions are translated

4.6.3 LDA.

The `lda` macro instruction loads addresses of variables. There is the potential for this instruction to be expanded two ways, depending on the type of the variable being addressed. Each form of the instruction has a more optimal compilation point for being expanded. We discuss each separately.

- The address of functions and global variables is obtained by loading a constant value. Therefore, lda can be expanded as ldc is expanded. The address of a function or global variable is not determinable at compile time, as the linker must add additional library and startup code. Instead, we have added annotations to each global variable

to indicate to the linker that the name should be replaced by the variable's address. An example of this expansion is shown.

```
lda r4, global_var   ==>  llo r4, LO(global_var)
                          lhi r4, HI(global_var)
```

Since the job of expanding these instructions is identical to that of ldc, expansion of lda instructions is also done during code generation.

- If the variable being addressed is a local variable or function parameter, it will be stored on the stack. Obtaining its address amounts to adding a constant offset to the stack pointer. An example is shown.

```
lda r4, local_var ==> addui r4, r31, offset
```

  The value of "offset" is the position of the variable or parameter on the stack. This position is not determined for variables until after the stack frame has been organized (section 4.3). Parameters to functions are not given stack positions until after code finalization. These two stages of compilation may also insert additional lda instructions. For these reasons, it is better to do macro expansion of `lda` instructions again after code finalization.

As these macroinstructions are not built into the ISA, the assembler or linker does not handle them. All macroinstructions must be expanded before compilation is complete. Macro expansion is performed after code finalization to ensure all macroinstructions are expanded into instructions suitable for the assembler.

4.7 SCMP Backend Summary

The SCMP backend turns the generic SUIF/MachSUIF compiler infrastructure into an SCMP specific compiler. The backend translates the various instructions of MachSUIF's SUIFvm instruction set into equivalent SCMP instructions.

Most compilers assume the existence of an operating system to perform such operations as network communication, memory allocation, and process initialization. In the SCMP architecture, we do not have any such operating system as of yet. Therefore, in regards to SCMP, all functionality generally reserved for an operating system must instead be implemented within the compiler.

Much of the design of the SCMP compiler is similar to that of other RISC architectures. The compiler must specify how each of the general-purpose registers is used (stack pointer, return address pointer, caller saved registers, etc.). The compiler must also decide how to organize a process' stack frame, which includes considerations for local variable and procedure argument storage. These issues relate to how the compiler arranges local procedure calls.

However, SCMP also permits remote procedure calls (RPCs), which involves stack initialization, rearranging of procedure arguments, and special procedure calls for returning results. Without an operating system, the compiler must insert code into a program to handle these issues.

Within the compiler, we have defined how remote procedure calls are made, how arguments are delivered to the procedure, and how completion notification and return values are sent. These definitions then influence the assignment of registers and the organization of the stack. In initializing the stack, the compiler must consider that, on each node, the remote procedure may be running on any of the node's contexts, and that it must ensure that each context has a separate segment of memory for its stack.

We find that not only must the compiler address issues common to most modern compilers, it must also partially fill the role of an operating system as well. We say partially, as some traditional responsibilities of common operating systems are handled by SCMP hardware (thread scheduling, network interfacing) while others are not necessary (virtual memory management, hardware management). While improvements are necessary to have a flexible and well-rounded compiler, we can safely establish that a compiler for the SCMP architecture is practical.

Chapter 5. Partial Redundancy Elimination

Partial redundancy elimination is a form of global common subexpression elimination. Common subexpression elimination attempts to remove a given expression if and only if it can guarantee that the result produced by the expression will be produced by an identical expression earlier in the program.

While the algorithm can remove any such redundancy, it is most effective at removing redundancies introduced by address calculations. Offsets into arrays or data structures will often produce many redundancies removable by some form of common subexpression elimination. Figure 5-1 shows an example of common subexpression elimination at work.

```
long i, n, *A, *B, *C, *M;
for(i=0; i<n; i++){
    A[i] = B[i] + C[i];
    M[i] = B[i] * C[i];
}
```

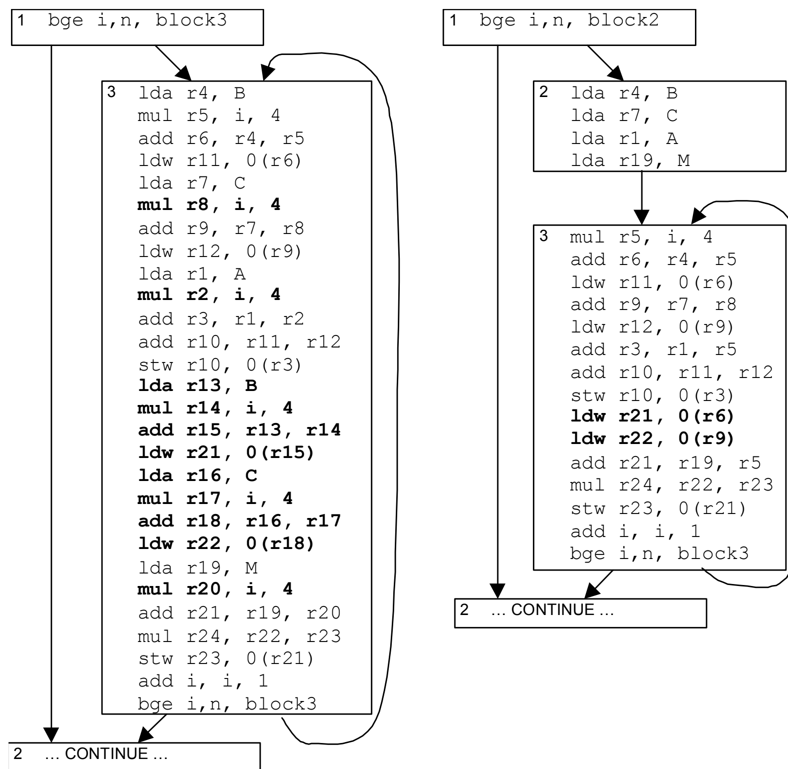


Figure 5-1. An example of a partial redundancy elimination.

Figure 5-1 shows a small portion of a C program. This segment of code can easily translate into almost 30 lines of unoptimized assembly code. The left column shows what the translated C code might look like as represented by unoptimized SUIF virtual machine (SUIFvm) instructions. Redundant calculations are shown in bold. The

compiler generates these redundancies for the purpose of address calculations. While a clever programming style can avoid these redundancies, a programmer should not need to worry with such detail. It is a better approach to remove these redundancies from within the compiler. The right column shows the power of common subexpression elimination, reducing the twenty-eight instruction intermediate code to nineteen instructions. With some of those instructions becoming loop-invariant (i.e. being moved outside of the loop) the size of the loop actually reduces from twenty-seven instructions to fourteen.

Notice that redundancies still exist in the optimized code of Figure 5-1. Additional instructions can be moved from the loop. These are all memory accesses. To move or remove these instructions, we must perform additional analysis or make several assumptions. Section 5.6 discusses an approach to removing memory access expressions.

As is evident, even the smallest program can have a great deal of redundancies, making common subexpression elimination a worthwhile investment. There are many algorithms for going about removing these redundancies. Some deal with basic blocks of code individually, yielding fast performance but with less effectiveness at removing redundancies. Other algorithms attempt to find redundancies over an entire procedure, considering all basic blocks when deciding which expressions are redundant. This is referred to as global common subexpression elimination.

Partial redundancy elimination takes this global approach to removing redundancies while adding a few other features. Introduced by E. Morel and C. Renvoise in 1976 [Morel1], the original algorithm seeks to integrate global common subexpression elimination with another optimization technique – loop invariant code motion – in a nearly linear set of data flow analyses and equations. The general idea of the algorithm is that it applies its equations to the control flow graph of a program, eliminating any computation if it has already occurred on certain control paths.

Several papers have been written on partial redundancy elimination since its inception, each hoping to improve upon the algorithm's effectiveness and performance. Knoop, Ruthing, and Steffen attempt to simplify the data flow analysis of the original algorithm [KRS1]. They also add the ability to reduce register pressure. Briggs and Cooper reveal limitations of partial redundancy and suggest applying the static single assignment (SSA) technique to simplify the implementation of partial redundancy and improve its performance [Briggs94]. Other proposals revise the partial redundancy algorithm using SSA, allowing optimization of memory accesses as well [Kenn1]. These papers then present results of their implementations on common benchmarks.

With all this research having been done, what is the value of another detailed discussion of partial redundancy elimination? Here are several reasons.

1. Many of the problems we find when implementing partial redundancy elimination are attributed to subtleties in the program code and in the intermediate format. These minute details are very difficult to overcome and largely overlooked by papers written on partial redundancy. How we handle these complications is worth an thorough discussion.

2. While some of the proposals concerning partial redundancy elimination are helpful, others prove to be detrimental to implementation, performance and effectiveness. (That is, the proposals make implementation more difficult, time to execute the algorithm becomes slower, and the algorithm is less able to make code improvements.) In Chapter 6, we provide some reasons why this is so.

3. Limitations imposed in some partial redundancy algorithms are easily avoided.

In this chapter, we examine our implementation of partial redundancy elimination, and compare it other algorithms. We attempt to reveal all of the obscure problems with the algorithm, show some limitations, and attempt to overcome these.

5.1 Lazy Code Motion

Lazy Code Motion is a partial redundancy technique proposed by Knoop, Ruthing, and Steffen [KRS1]. Lazy code motion attempts to resolve the original series of equations in partial redundancy elimination into a simpler set of equations. These equations are intended to make the algorithm more efficient. Lazy code motion also avoids moving expressions away from their uses. This reduces the need for registers, potentially reducing the need for spilling during register allocation.

Lazy code motion depends upon two sets of local data flow analyses. The analyses determine if each occurrence of each expression has either of two properties: local transparency and local anticipatability. These properties are considered local as they only depend on other instructions within the same basic block. We discuss the evaluation of these properties later.

- Local Transparency (*TRANSloc*). An expression is locally transparent in a basic block if there are no assignments in the block to variables in the expression.

- Local Anticipatability (*ANTloc*). An expression is locally anticipatable in a basic block if the expression occurs in the block and the effect of the block is unchanged when the expression is moved to the beginning of the block.

With these properties established, we can evaluate the remaining data flow equations for lazy code motion. The remaining equations for lazy code motion are shown below.

$$ANTin(i) = ANTloc(i) \cup \left(TRANSloc(i) \cap ANTout(i)\right)$$
$$ANTout(i) = \bigcap_{j \in Succ(i)} ANTin(j)$$
$$ANTout(exit) = \varnothing$$

Equation 5-1. Data flow equations for global anticipatibility.

$$EARLin(entry) = \bigcup \exp$$
$$EARLin(i) = \bigcup_{j \in Pred(i)} EARLout(j)$$
$$EARLout(i) = \overline{TRANSloc(i)} \cup \left(\overline{ANTin(i)} \cap EARLin(i)\right)$$

Equation 5-2. Data flow equations for earliestness.

$$DELAYin(entry) = ANTin(entry) \cap EARLin(entry)$$
$$DELAYout(i) = \overline{ANTloc(i)} \cap DELAYin(i)$$
$$DELAYin(i) = \left(ANTin(i) \cap EARLin(i)\right) \cup \bigcap_{j \in Pred(i)} DELAYout(j)$$

Equation 5-3. Data flow equations for delayedness

$$LATEin(i) = DELAYin(i) \cap \left(ANTloc(i) \cup \overline{\bigcap_{j \in Sicc(i)} DELAYin(j)} \right)$$
$$ISOLin(i) = LATEin(i) \cup \left(\overline{ANTloc(i)} \cap ISOLout(i)\right)$$
$$ISOLout(i) = \bigcap_{j \in Succ(i)} ISOLin(j)$$
$$ISOLout(exit) = 0$$

Equation 5-4. Data flow equations for isolated expressions

$$OPT(i) = LATEin(i) \cap \overline{ISOLout(i)}$$
$$REDN(i) = ANTloc(i) \cap \overline{LATEin(i) \cup ISOLout(i)}$$

Equation 5-5. Final results for optimal and redundant expressions.

Lazy code motion is convenient in that its performance is easily controlled. If we give two identical instructions different expression names, we prevent lazy code motion from combining them. When certain variables require special consideration, we can modify our local transparency and local anticipatability analyses to accommodate these variables. We find this flexibility helpful when performing alias analysis. In either event, we have the power of making significant behavioral modifications to the algorithm without modifying the equations shown above. Improvements to the effectiveness of lazy code motion can be made easily without changing the algorithm.

5.2 Control Flow Graph Properties

An optimizing algorithm may require that a control flow graph have certain properties if the algorithm is to be effective. Some of these graph transformations can be incorporated into the algorithm, allowing the algorithm to operate effectively despite how previous transformations modify the graph. Other graph transformations must be done in earlier compilation stages. These transformations must be done separately requiring that the

algorithm rely upon previous optimizations for effective operation. For partial redundancy elimination to work effectively, it requires certain control flow graph properties falling into both of these categories.

5.2.1 Control Flow Graph Introduction

A control flow graph represents the different paths of execution within a procedure. The directed graph consists of basic blocks and edges. Each basic block represents a sequence of instructions. A branch instruction may be included as the last instruction in the block.

The edges of a control flow graph represent potential paths of execution among basic blocks. We define a path as flowing from the tail of an edge to its head. For the graph shown, there are two edges. Block 1 is described as being at the tail of each edge.

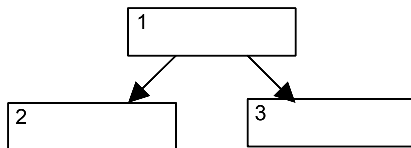


Figure 5-2. A sample control flow graph.

Edges also define the predecessors and successors of a block. A block's predecessors consist of all blocks that may immediately precede the block. In our example, blocks 2 and 3 have as their only predecessor block 1. Successors are the blocks that may follow a certain block. In our example, block 1 has two successors: blocks 2 and 3. A block may have two or more successors if the last instruction of the block as a conditional branch. Otherwise, the block may have only one successor.

Many documents use the term nodes in place of basic blocks in a control flow graph. We avoid this usage, as it might be confused with the concept of the SCMP node.

5.2.2 Critical edge splitting

For full effectiveness of partial redundancy elimination, we must incorporate a commonly used technique of graph transformation referred to as critical edge splitting. A critical edge is defined as an edge whose tail has multiple successors and whose head has multiple predecessors. Critical edges can prevent code motion.

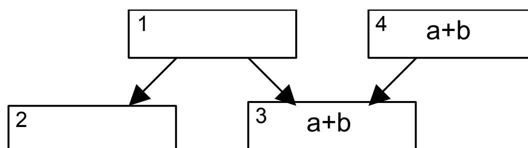


Figure 5-3. Optimization limited by a critical edge.

In Figure 5-3, the computation "a+b" at block 3 is partially redundant. However, since an execution path exists from block 1 to block 2 avoiding the computation in block 3, partial redundancy cannot move the computation of "a+b" from node 3 to node 1. By splitting the critical edge (1,3), i.e., inserting node 5 along the edge, partial redundancy is able to remove the redundant computation. The resulting graph and code motion is shown in Figure 5-4.

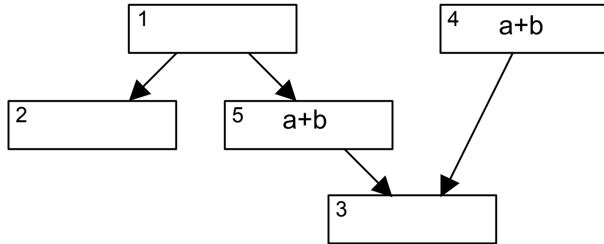


Figure 5-4. Transformed graph permits code motion.

The lazy code motion algorithm requires that any edge leading to a node with multiple predecessors be split. This additional requirement simplifies data flow analysis and code placement techniques. As edge splitting must be done prior to each pass of the algorithm (section 5.5), we do not need to ensure that a control flow graph has its critical edges spilt before applying partial redundancy elimination. This transformation must be incorporated into our algorithm.

5.2.3 Critical Block Splitting

An issue with partial redundancy not addressed in any previous algorithm pertains to the subject of critical blocks. A critical block is a basic block with multiple predecessors and multiple successors. Like critical edges, they inhibit full effectiveness of many optimizing algorithms. Unlike critical edges, critical blocks cannot be split as easily. An optimization pass must rely upon the compiler to avoid generation of such nodes.

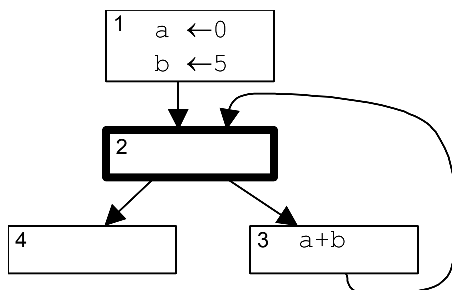


Figure 5-5. A critical node preventing code motion.

Observe the graph in Figure 5-5. The expression "a+b" appears to be loop invariant. However, if we move the expression to block 1, it will execute even if the loop is never entered. There is no way to transform the critical node (block 2) so that it allows such movement. Therefore, the expression "a+b" cannot be removed and the redundancy remains.

Since we cannot remedy critical blocks, we must avoid them. To avoid them, we must know why they are created. Critical edges are generated during the translation from the lowering of the SUIF intermediate language. Critical blocks come about given one of two circumstances.

1. Translation of loops.

   SUIF generates a critical block whenever a loop is lowered into tests and branches. Such loops include C's for and do/while loops and FORTRAN's do loops. Consider the example shown. The sample C code segment below is translated into a single test at the head of the loop and a jump to the head from the tail.

```
x = -20;
for (; x<0; x++){
  c = a+b;
}
print(x);
```

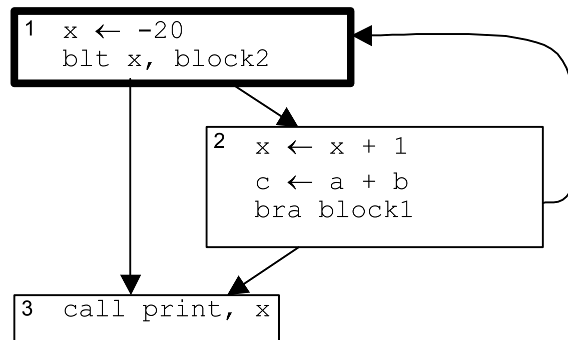


Figure 5-6. The critical node prevents any code motion

By giving the loop an entry test at the head of the loop and a continuing test at the loop's tail, the critical block is replaced by a critical edge. The critical edge can then be split, allowing redundancy elimination. Resulting optimizations on the transformed graph are shown in Figure 5-7. Other loop constructs can be translated similarly.

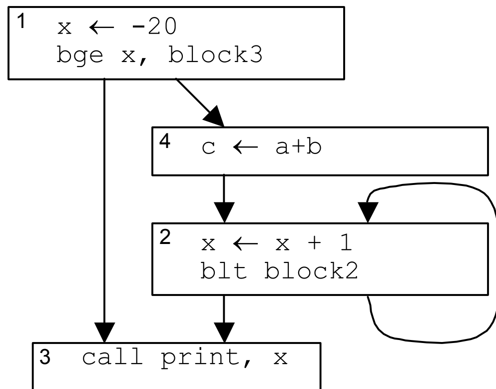


Figure 5-7. Creating critical edges instead of critical nodes allows code motion.

2. Consecutive tests and branches.

   SUIF also generates critical blocks wherever consecutive tests and branches are translated. Consider the example shown in Figure 5-8. These critical blocks do not present a problem with partial redundancy elimination. Since the expressions in our example cannot be moved out of the conditional branch, they cannot be moved from any enclosing loop. The partial redundancy algorithm would therefore leave the expression inside the loop. Critical nodes generated by consecutive tests and branches can be left in the control flow graph without inhibiting partial redundancy elimination.

```
for( x=0; x<20;){
  if( y < 0 )
    z += y;
}
```

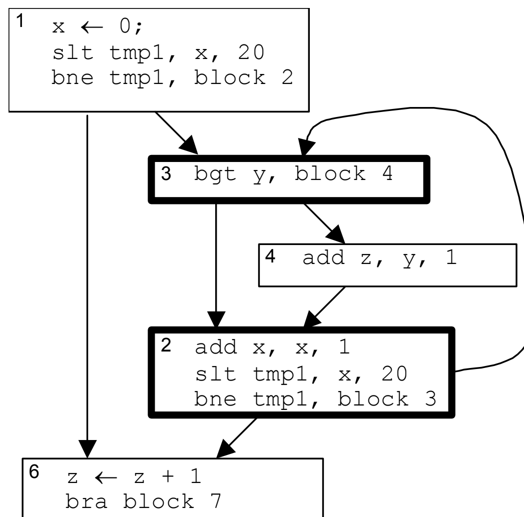


Figure 5-8. Critical nodes caused by test and branch translation.

These conditions are SUIF specific, but must be avoided by any compiler infrastructure if partial redundancy elimination is to work effectively.

5.3 Expression Naming

The first challenge to overcome in implementing lazy code motion lies in identifying expressions. For any expression to be deemed redundant it must be identified as being the same as some other expression that precedes it, i.e., the expressions must have the same name. Naming schemes are not addressed in either the original partial redundancy paper, or in the paper presenting lazy code motion. However, expression naming deserves some discussion, as it can hide potential optimizations from our algorithm if done poorly.

Before creating an algorithm for naming expressions, we have to define several rules for how expressions can be optimized. Certain instructions cannot be handled by lazy code motion. These instructions must be untouched, even if lazy code motion can eliminate them or combine them with other instructions. Here are the rules for handling special instructions:

- Any memory access expression cannot be handled by lazy code motion. Memory access expressions include any instruction having a referenced variable (a variable with its address taken), a global variable, or a volatile variable as a source or destination. A detailed discussion of this restriction is presented in section 5.6 when we discuss an algorithm for overcoming it.

- Any two expressions cannot have the same name if they have different destinations and either destination is the destination of another instruction. The reason for this rule is shown in figure 5-9. If the add instructions in node 1 and node 2 are given the same name, lazy code motion may produce the result shown.

```
b = 0;
a = m + n;
if( m < n )
  b = m + n;
c = a + b;
```

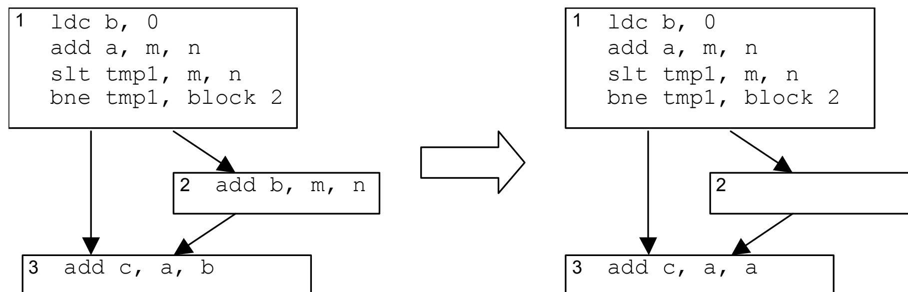


Figure 5-9. Lazy Code Motion incorrectly removing a required expression.

Fortunately, instructions with these multiply assigned variables need not be entirely disregarded. Figure 5-10 shows a program in which lazy code motion can transform the graph to produce a more optimal result.

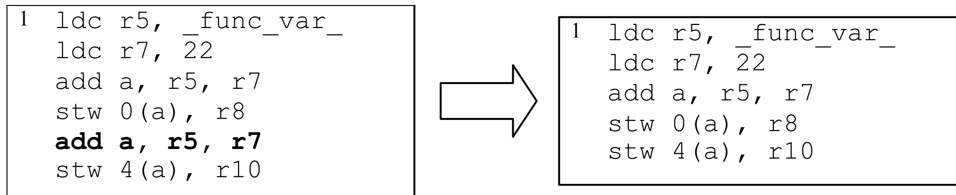


Figure 5-10. Lazy code motion moving an expression correctly.

> This condition may seem obvious and avoidable. It is specifically handled due to the unpredictability of intermediate code transformations. An earlier compiler transformation may not be sufficiently intelligent to detect the redundancy. Such circumstances are left to our optimization pass, and we must handle the condition.

We begin expression naming by establishing a record of named expressions. We call this the *expression catalog*. The expression catalog maintains each expression's name and a list of instructions that have been given that name. When determining the expression name of a new instruction, we compare it to each entry in the expression catalog. Since an expression's name must be unique (an instruction cannot match two entries in the catalog), any match found determines the instruction's expression name. If no match can be found in the catalog, the instruction can then receive a new expression name. That name must then be added to the catalog.

The difficulty then becomes how to name instructions so that each can receive only one expression name. The expression name given that instruction determines what, if any, instructions evaluate the same expression.

Then, our algorithm for assigning expressions names to instructions goes as follows.

```
for each instruction in the procedure
    if the instruction is a memory access instruction, it must be given its own
      expression name and entry in the catalog. By giving each memory access
      instruction a different name, we prevent any memory access expressions
      from being moved or removed by partial redundancy elimination.
    else, for each entry in the expression catalog
        if the operation of the expression does not match that of the catalog entry
          (add, sub, mul, etc.), the expression does not match the entry. Examine
          the next catalog entry.
        else if the expression and entry have a different number of source
          operands, the expression does not match the catalog entry. Examine the
          next catalog entry.
        else compare the sources of the expression to the entry (first source to first
          source, second source to second source, and so on).
          if the two instructions do not have the same sources in the same order
            if the operation is not commutative
              examine the next catalog entry
            else the operation is commutative,
```

compare the sources of the expression to the sources of the entry, reversing the order of the expression's sources (last source to first source, second to last source to second source, and so on). If the sources do not match, the expression does not match the catalog entry. Examine the next catalog entry.

else if the expression or the entry has a destination that is assigned multiple times, the destination of the expression must match that of the entry. If they do not match, the expression must have a new name.

else the expression matches the entry. The two instructions can be given the same expression name.

if after examining all entries in the catalog no match is found, the instruction receives a new expression name. That name is then added to the expression catalog.

5.4 Multiple instructions within a basic block.

All binary arithmetic for partial redundancy elimination can be performed on basic blocks as well as individual instructions. With multiple instructions allowed to exist in each basic block, the execution time for the algorithm can drop dramatically. Studies show that average basic block size is between eight and twelve instructions [Patel00]. Many heavily used functions may have only one basic block containing several hundred instructions. Partial redundancy can see on average a speedup factor of 4 to 8 in the average case, while functions with only several basic blocks can take nearly constant time. Optimization algorithms can benefit greatly by operating on these basic blocks instead of operating on individual instructions.

However, problems exist when applying partial redundancy to multiple instructions simultaneously. Given the lazy code motion algorithm as described by Knoop, Ruthing, and Steffen, lazy code motion can make invalid graph transformations if multiple instructions exist within a basic block. To avoid these invalid transformations, excessive restrictions must be imposed on the algorithm, making it much less effective.

Figure 5-11 shows a basic block with the same expression occurring twice in a basic block. Beside each instruction is an expression identifier. The first instance of expression A is locally anticipatable, while the second is not. Given that partial redundancy only records information about the entire basic block, it must either record that expression A is locally anticipatable or that it is not locally anticipatable for the entire block. This example shows how partial redundancy behaves if it considers expression A to be locally anticipatable in the block.

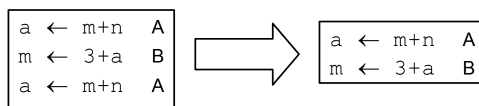


Figure 5-11. Partial redundancy elimination with multiple instructions in a node.

To avoid this problem, we must mandate that if an expression appears more than once in a basic block, and one of the computations of the expression is not locally anticipatable, then none of the computations can be locally anticipatable. If an expression cannot be locally anticipatable, it cannot be redundant, and therefore cannot be moved or removed. While this modification solves the problem, it can be a severe limitation on the algorithm. The loss of effectiveness is especially severe within large basic blocks. Figure 5-12 shows the potential loss of optimization due to this limitation.

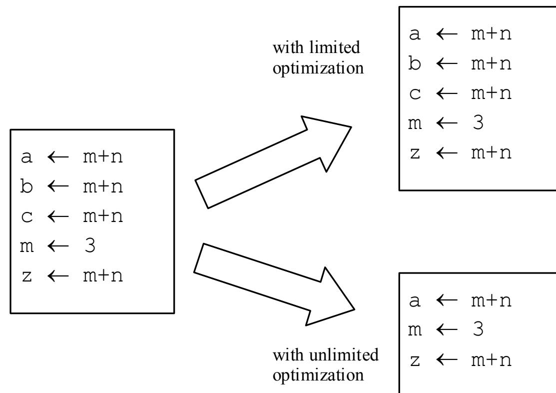


Figure 5-12. Ineffective partial redundancy with multiple instructions in a block

Partial redundancy's complications here stem from its basic design. Algorithms perform data flow analysis between basic blocks after performing initial data flow analyses within each block. The results indicate expressions that should be added or removed from each basic block. Morel and Renvoise suggest placing these expressions at the exit of basic blocks. Knoop, Ruthing, and Steffen limit insertion of instructions by lazy code motion to basic block entry. Inserting instructions at basic block exits is possible, but it requires modifications to the lazy code motion algorithm. No mention is made of how expressions should properly be removed.

To see if we can avoid limitations on lazy code motion through careful expression placement and removal, consider the series of instructions in figure 5-13. The redundant computations of expression A are italicized; the required computations are in bold. In this example, there are three required computations. Each required computation is followed by a redundant computation and an assignment to a variable of the expression. These assignments cause the following computations of the expression to change, thus requiring those computations.

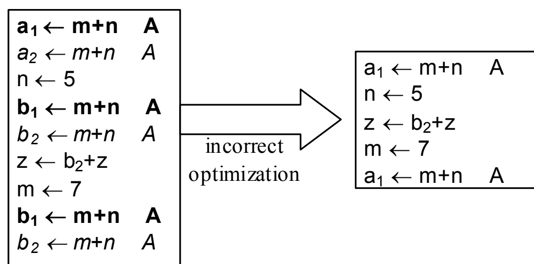


Figure 5-13. Redundant expressions can't be placed only at the entry and exit of a basic block.

Even if instructions are placed at both the entry and exit of this basic block, the block can not be optimized properly. During local availability calculations, expression A appears to be both locally available and not locally available in three different sections of the block. By simply placing computations at the entry and exit and removing all others, we remove necessary computations of expression A in the middle of the block.

The most effective, though least desirable solution is to apply the binary equations of lazy code motion to each instruction individually. The control flow graph libraries provided by MachSUIF allow us to transform a graph so that each instruction has its own basic block. Dataflow analysis can then proceed as before. The benefits of our set arithmetic shown initially can still be realized, as we do not need to analyze each instruction beyond the initial local anticipatability and local transparency analyses. The main disadvantage to this approach is that our set arithmetic now requires a larger set (one for each instruction as opposed to one for each block).

Optimization across basic blocks.

There is some disagreement as to whether partial redundancy elimination algorithms can move expressions across basic blocks. According to the previous discussion, one might assume that partial redundancy *must* be able to move expressions across basic blocks. However, Briggs and Cooper state that "an expression defined in one basic block may not be referenced in another basic block" when referring to partial redundancy [Briggs94]. The example they use is shown in Figure 5-14.

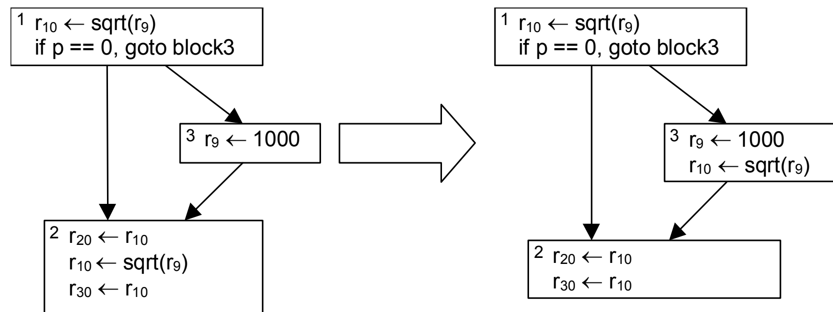


Figure 5-14. Invalid movement of an expression across a basic block boundary.

The lazy code motion algorithm solves the issue of code hoisting by modifying the local anticipatable property of expressions. Morel and Renvoise state that an expression is locally anticipatable when "there is at least one computation of the expression in the block *i*, and if the commands appearing in the block before the first computation of the expression do not modify its operands" [Morel1]. Lazy code motion appends to this definition a stipulation on uses of the expression. Muchnik states that an expression is locally anticipatable "if there is a computation of an expression in the block and if moving that computation to the beginning of the block would leave the effect of the block unchanged, i.e., there are neither uses of the expression nor assignments to its variables in the block ahead of the computation under consideration." [Much97]

By including the destination of an expression in performing data flow analysis, and by using the lazy code motion's definition of local anticipatability, a partial redundancy algorithm can avoid the problems mentioned by Briggs and Cooper. This allows expressions to be moved across basic blocks.

5.5 Local Data Flow Analysis

With expressions identified, we perform the initial local dataflow analysis. For each basic block, we must determine what expressions are locally transparent and what expressions are locally anticipatable.

5.5.1 Locally Transparent

As stated earlier, an expression is locally transparent in a basic block if there are no assignments in the block to variables in the expression.

With only one instruction in each block, we begin local transparency analysis by determining what operands are used as destinations in a block. With that determined, it is a simple task to search all expressions for instances of those operands. Any expression containing an operand that is a destination in the block cannot be locally transparent. When searching for these destination operands, we must remember to check all sources and destinations in each expression. Otherwise, partial redundancy will perform such incorrect code motion as is shown earlier in Figure 5-14.

When defining expressions, we always give a memory access instruction its own expression name. This is to prevent it from being considered redundant. However, the instruction may still be considered loop invariant. This can lead to invalid code transformations. We prevent this code motion by declaring all memory access expressions not locally transparent in any block where there is a write to memory. In other words, if a block contains a memory write instruction, or it contains a destination operand that is referenced or is global, then all memory access expressions are not locally transparent in that block. The memory access expression that is contained in a block is always considered locally transparent.

Procedure calls also present a dilemma for memory access expressions. As we do not know if a procedure modifies a variable or memory location, we must assume that it does. Consider the following example.

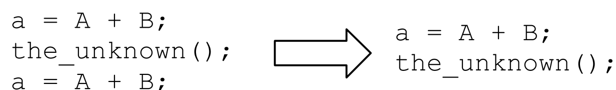


Figure 5-15. Lazy code motion at work where perhaps it shouldn't be.

Lazy code motion without regard to memory access would effectively optimize the second addition of figure 5-15 away. If A and B are local, and neither has its address taken, the second addition and store is redundant and all is well. But if either A or B has its address taken, the_unknown( ) may modify the variable before returning, making the

second addition necessary. Since we know nothing about the procedure, we can only determine that the second computation *may not be* redundant. To prevent its movement past a procedure call, we make all memory access expressions not locally transparent in any block that contains a procedure call.

5.5.2 Locally Anticipatable

An expression is locally anticipatable in a basic block if the expression occurs in the block and the effect of the block is unchanged when the expression is moved to the beginning of the block. Since our analysis only allows one instruction per block, this analysis is simple. If an expression exists in the block, it is locally anticipatable. All other expressions are not locally anticipatable in the block.

5.5.3 Dependant Dependencies

Partial redundancy elimination relies heavily upon the correct determination of expression names for effective optimization. It must be able to name multiple instructions with the same expression name before an instruction can be removed. However, many redundant instructions will not receive the same expression name due to dependencies on other redundant instructions.

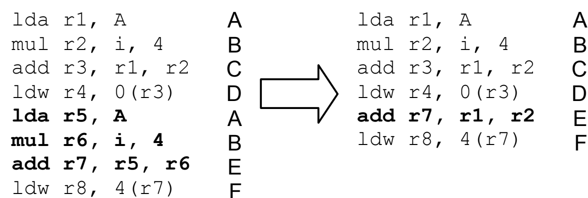


Figure 5-16. Partial redundancy is not able to remove redundant expressions until they receive the same name. One pass of the algorithm may not be sufficient.

In Figure 5-16, we see the effects of partial redundancy elimination after one pass of the algorithm. The expression name assigned to each instruction is shown. Notice that there is only one instruction with the expression name “E”, yet it is redundant. It is the same computation as expression “C”. Because it depends on the redundant computations of expression “A” and “B”, our naming scheme cannot identify expression “E” as being the same as expression “C”. It therefore cannot remove the redundant computation.

We resolve this issue by making multiple passes of the partial redundancy algorithm on program code. In our example, the remaining redundancy is exposed after our expression-naming scheme is applied on the second pass. Of course we have no way of knowing if two applications of the algorithm is sufficient to remove all redundancies. We must continue to apply our algorithm until no changes occur. In this example, three passes are required.

```
lda r1, A        A
mul r2, i, 4     B
add r3, r1, r2   C
ldw r4, 0(r3)    D
lda r5, A        A
mul r6, i, 4     B
add r7, r5, r6   E
ldw r8, 4(r7)    F
```

First Pass

```
lda r1, A        A
mul r2, i, 4     B
add r3, r1, r2   C
ldw r4, 0(r3)    D
add r7, r1, r2   C
ldw r8, 4(r7)    F
```

Second Pass

```
lda r1, A        A
mul r2, i, 4     B
add r3, r1, r2   C
ldw r4, 0(r3)    D
ldw r8, 4(r3)    F
```

Third Pass

```
lda r1, A        A
mul r2, i, 4     B
add r3, r1, r2   C
ldw r4, 0(r3)    D
ldw r8, 4(r3)    F
```

No Change

Figure 5-17. To remove all possible redundancies, we must apply partial redundancy elimination until no change occurs.

5.6 Moving and removing memory access expressions.

One assumption made in our initial implementation of lazy code motion was that we could not move any expression that requires a memory access. We can find some additional benefit by allowing motion of these memory accesses.

In Figure 5-1, the values of B[i] and C[i] must be loaded twice, even after partial redundancy elimination is applied, as memory accesses are involved. Assuming arrays B and C are modified only by this program, the second load from each array is redundant. In Figure 5-18, we find that the redundant memory accesses are removed, eliminating two instructions from the loop.

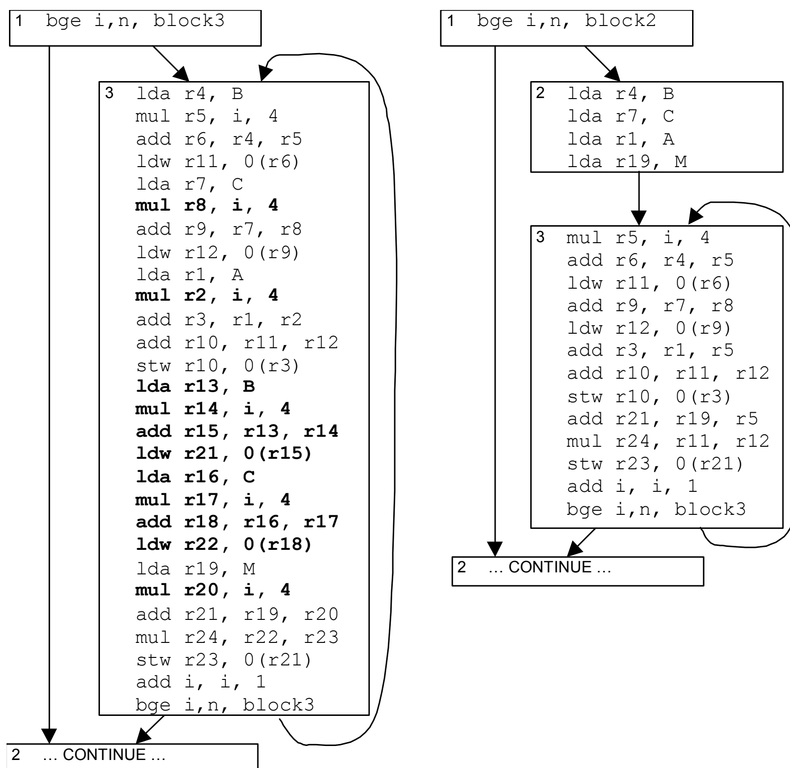


Figure 5-18. Potential redundancies in memory access expressions.

While the ability to move memory access expressions is beneficial, there will continue to be limitations on what expressions can be moved. Without the ability to do

interprocedural analysis, we cannot know how functions might be accessing global and address taken variables. In a multithreaded environment, other threads modify variables. Given that programs must operate in these conditions, we are faced with circumstances in which redundancies are difficult to isolate.

To determine the rules for when a memory access expression can be moved, we need to examine our justification for originally decided to not move them. Remember that an expression is considered to access memory when either it is a load or a store, or when any of the expression's operands, whether it is a source or a destination, is global or has its address taken.

Consider the example in Figure 5-18. Our goal for simplifying memory access expressions is to remove the remaining load instructions in block 3. Assuming arrays A and B address separate portions of memory, these loads are redundant. However, a more complete example as in figure 5-19 reveals a problem. Let the C code in figure 5-1 be part of the code presented below.

```
main(){
    long i, *A, *B, *C, *M;
    A = init();
    B = init();
    C = init();
    M = init();

    for( i=0; i<size(); i++){
        A[i] = B[i] + C[i];
        M[i] = B[i] * C[i];
    }
}
```

Figure 5-19. Complete program using code of figure 5-1.

Note here that all pointers are initialized with unknown values. If arrays A, B, and C are initialized to the same block of memory, B and C must be reloaded after computing the first expression, as that computation will affect the value of the second expression. From this example, we can determine that a memory access expression cannot be redundant if it follows a store to an unknown address.

Considerations must be made beyond these for moving memory access expressions. SCMP is designed as a multiprocessing, multithreaded environment. Any optimizer must consider multithreaded applications when deciding how aggressive it can be in dealing with expressions. As C and most other programming languages do not have built-in support for multithreaded applications, the compiler has no direct way of determining data-dependencies between threads of execution.

```
A = 1;
B = 1;
start_remote(&A);
while(A || B){ /* the waiting loop */
      B = A;
} /* wait until A=0 and B=0 */
```

Figure 5-20. Code segment of a multithreaded application

Figure 5-20 shows the problem. Here, the programmer intends for the program to stay in the waiting loop until A is set to 0. Then the loop is executed once more so that B is also set to 0. The programmer has designed some other thread that will modify A as necessary. However, lazy code motion regards B = A to be loop invariant and will effectively produce the code in figure 5-21.

```
A = 1;
B = 1;
start_remote(&A);
B = A;
while(A || B){ /* the EMPTY waiting loop */
} /* waiting forever */
```

Figure 5-21. A corrupted multithreaded application.

In this case, B will likely never be set to 0, and the programmer will have a waiting loop that will wait forever. Without some information from the programmer, this becomes an unsolvable problem. We must either have the programmer provide some information about the external use of the variable or abandon optimization of memory access expressions.

Fortunately, there is a solution that has gained general acceptance, though it is not entirely part of all language standards. The `volatile` keyword in C can be used to indicate those variables that are untouchable by optimizers. Volatile variables are modified outside the current program and no amount of data flow analysis can be applied to correctly determine how they are used.

For our example in figure 5-20, by making A volatile, the optimizer understands that the variable is modified externally to the program, preventing the statement B = A from being loop invariant.

Now that we know how to move memory access expressions, here are our revised rules for moving them:

1.) No expression containing a volatile operand can be moved outside its basic block, nor can it be removed, even if an identical expression exists in the same basic block.
2.) No expression containing an address expression that addresses a volatile operand can be moved outside its basic block, nor can it be removed even if an identical expression exists in the same basic block.

5.6.1 Expression Naming

In our original expression naming scheme, we prevented the removal of memory access instructions by giving each its own expression name. Since we now want to allow the removal of these instructions, we must modify our expression-naming scheme. We do this by removing the earlier constraint on memory access instructions. Memory access instructions can now be compared during expression naming just as we compare other instructions.

5.6.2 Local Data Flow Analysis

It is the job of our local data flow analysis to perform the alias analysis between variables. Several problems arise when attempting to perform this analysis. We find that multiple types of instructions modify a variable when it is stored in memory, preventing proper dataflow analysis. Figure 5-22 shows an example program and its MachSUIF representation. Partial redundancy moves instructions from the loop even though they are not redundant. The problem arises because the variable *ga* can be accessed either directly or via a pointer. Since we cannot determine the dependencies between these instructions, our partial redundancy algorithm views both as being loop invariant.

```
int ga;
int init(){ return &ga; }
int proc1(){
    int *lp = init();
    ga = 3;
    while( (*lp) > 0 ){
        (*lp) = ga + 1
    }
    return ga;
}
```

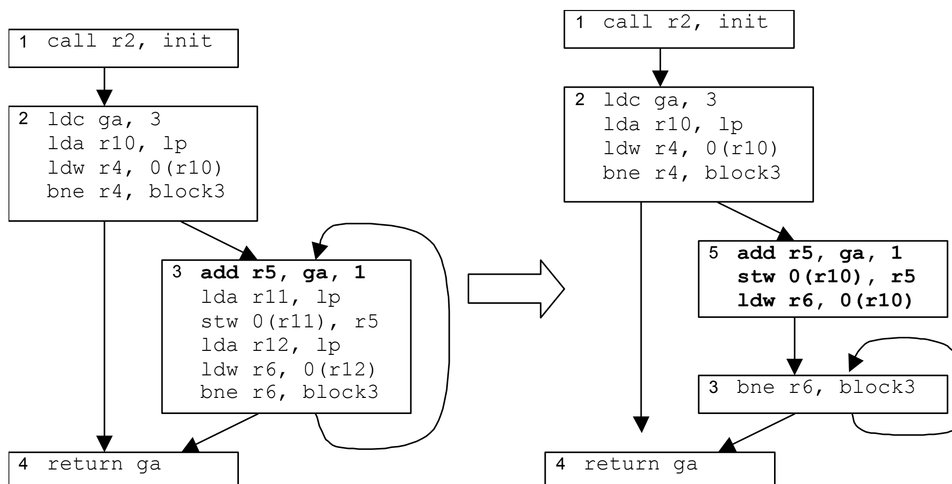


Figure 5-22. Incorrect redundancy elimination on memory access expressions. Notice movement of expression A from the loop.

When performing data flow analysis, we must consider that memory loads and stores may interact with instructions that contain referenced or global variables. How these various instructions interact is not always determinable. In such cases, we must assume that the instructions interact.

Note that while a global variable cannot be accessed via a memory reference if the variable's address is not taken, we must still assume the variable is referenced. A global variable can be used outside of the file in which it is declared, making it possible for the variable to be referenced even though it may not appear so in our intermediate code.

These issues are avoided if we can make one assumption: if a memory access expression writes to its memory location, no other expression accesses that location with a different variable or pointer. This is not the case when two pointers are assigned the same value, or when a pointer is assigned the value of a global variable. The procedures shown in Figure 5-23 are examples where we cannot make our simplifying assumption. However, it is far more common that the assumption can be made, giving us more freedom in moving memory access expressions.

```
int ga;
int proc1(){
    int *lp = ga;
    ga = 3;
    (*lp) = 4;
    return ga;
}
```

```
int proc2(){
    int la = 0;
    int *a = &la;
    la++;
    (*a) = 5;
    return la;
}
```

```
int proc3(){
    int la = 0;
    int *a = &la;
    la++;
    (*a) = 5;
    return la;
}
```

Figure 5-23. Procedures that allow access by multiple variables to a common memory location. Memory access optimizations may not be possible with partial redundancy elimination.

With the simplified alias analysis, all special handling of memory access expressions is handled through the naming of expressions. No changes are required to the locally anticipatable or locally transparent calculations. Unfortunately, we cannot make a sufficient determination of whether this property holds true from within the compiler. The programmer must supply this information to the compiler by way of command-line options.

5.7 Partial Redundancy Conclusions

While the Lazy Code Motion algorithm designed to remove partially redundant instructions is simple in concept, it is difficult to implement in practice. Many important details are avoided in papers written about Lazy Code Motion. Perhaps this important information is avoided to simplify discussion of the algorithm. However, these details become critical when putting lazy code motion into practice. The following are considerations we made in implementing Lazy Code Motion.

- Critical Block splitting

Previous partial redundancy algorithms emphasize the splitting of critical nodes. However, none mention the breaking of critical blocks, which also prevent code motion. Our research has shown that these critical blocks cannot be split in the same manner as critical nodes. Instead, their generation must be avoided during higher levels of compilation.

- Expression naming

  An instruction's name determines what expression it evaluates. While traditional algorithms have examined only the source operands and the operator of an instruction when determining the expression it evaluate (and thus, it's expression name), we find that in real applications, any destination operands included in the instruction are also important. Two instructions that have the same sources and the same operator may also have different destinations, of which either may be assigned elsewhere in a program, making the two instructions not necessarily redundant.

- Basic Block size

  Lazy Code Motion operates on basic blocks, not on individual instructions. This mode of operation allows any number of non-branching instructions to be contained within each block, allowing greater efficiency when executing the algorithm. However, we find that problems are incurred when several instructions appear redundant within a block but are in fact necessary. By allowing only one instruction in each basic block, we alleviate this problem without making changes to the original Lazy Code Motion algorithm. We also simplify the local data flow analysis.

- Local data flow analysis

  Lazy code motion provides for two forms of initial data flow analysis: locally anticipatable and locally transparent. Locally anticipatable analysis of an instruction depends on previous instructions within the same block. Given that we allow only one instruction in each basic block, this analysis is moot, simplifying our local data flow analysis to local transparency.

- Memory Access Expressions

  Partial redundancy elimination algorithms also do not address memory access expressions when removing redundancies. This is due in part to the high degree of difficulty in determining dependencies between memory accesses. However, in practice, dependencies between memory accesses can often be overlooked during optimizations if the compiler can be informed of when to overlook these potential (but unknown) dependencies. We can easily design our compiler so that the programmer can supply this information, greatly improving the optimization performance of our lazy code motion algorithm.

These considerations lead to a complete, robust, and effective algorithm for removing partial redundancies. Testing results of the algorithm are shown in Chapter 8.

Chapter 6. Optimizing with Static Single Assignments

Static single assignment (SSA) form is a relatively new approach to developing optimization algorithms. Its goal is to incorporate data flow information into a control flow graph, making algorithms simpler to develop and implement, and to allow these algorithms greater effectiveness in performing optimizations. However, the SSA form can be the source of many complications when used to implement certain algorithms. In this chapter, we present some of these difficulties. We use the lazy code motion algorithm implemented in Chapter 5 as an example.

6.1 SSA Introduction.

Before discussing the difficulties of using SSA form, we present a brief introduction to the SSA technique. This introduction is not intended to be a detailed description of the SSA form. For a more thorough description of SSA, consult *Static Single Assignment Construction* [Briggs98], *An Efficient Method of Computing Static Single Assignment Form* [Cytron89], or *Advanced Compiler Design and Implementation* [Much97].

The foundation of the SSA form lies in the fact that each variable in the control flow graph is given a unique SSA name for each instance in which the variable is assigned. Therefore, if two instructions assign the same variable, that variable is given two SSA names. The SSA names are then used in place of the original variable names. Figure 6-1 shows a sample control flow graph and its equivalent SSA graph.

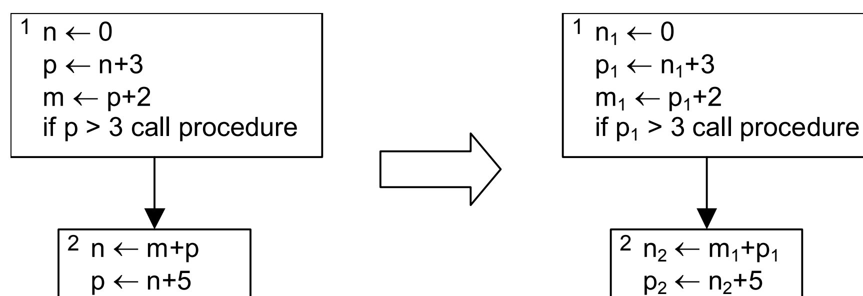


Figure 6-1. A control flow graph and its SSA equivalent.

With more complex graphs, two assignments to a variable may be valid at a basic block. The PHI instruction is introduced to merge multiple valid SSA names in the node. It has two or more sources, each source representing an assignment. One of the sources is *assigned* to the destination SSA name of the PHI instruction. The new SSA name is then used in place of the source SSA names along any succeeding control path.

Of course, the PHI instruction has no real implementation. The sources of the PHI instruction are meant only to represent the possible assignments that could reach the basic block while maintaining the necessary property of static single assignment form – an SSA name is assigned only once. (Many papers written on the SSA form refer to this as

a PHI node. We use the term PHI instruction so as to more easily distinguish it from a CFG node or basic block.)

Figure 6-2 shows a graph requiring a PHI instruction. In the CFG, there are assignments to variable m in blocks 1 and 3. Either assignment can reach block 2, depending on whether the branch is taken to block 3. The PHI instruction in block 2 represents the merging of two potential values. The result is assigned a new SSA name for variable m.

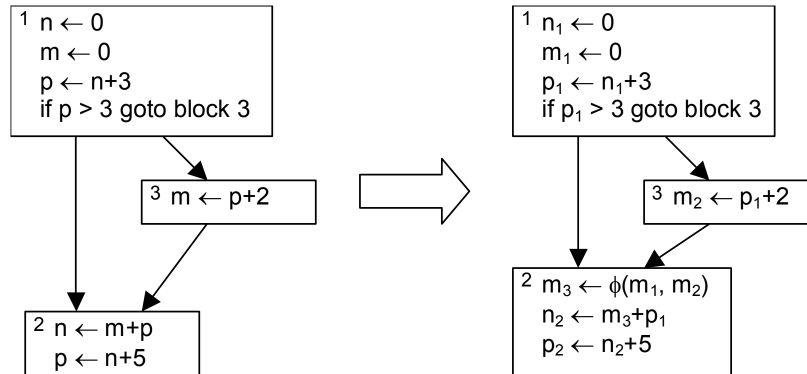


Figure 6-2. SSA graph needing a PHI node.

Optimization algorithms that wish to use SSA can convert from a regular control flow graph to SSA form, perform transformations on the SSA graph, and convert the transformed graph back to CFG form for further processing. It is possible to transform the SSA graph so that it is no longer valid. We discuss this dilemma in section 6.2.3.

MachSUIF provides SSA libraries for transforming a control flow graph to SSA form and back again [Holl2]. The libraries also provide utilities for making transformations to the SSA graph. The design of the libraries is based on the algorithms *Static Single Assignment Construction* developed at Rice University [Briggs98].

6.2 Partial Redundancy Using the SSA Form.

Several papers have suggested using the static single assignment (SSA) form of a control flow graph when implementing partial redundancy elimination. The intention of using SSA is to make the partial redundancy algorithm simpler to implement and cheaper to execute while not affecting its ability to optimize.

In our implementation of partial redundancy, we compare two implementations of the algorithm: one using the original CFG, the other using the SSA CFG. In our comparison, we examine such things as processing time of the lazy code motion algorithm and simplicity in design. Our concern is that the SSA form of an optimization algorithm can actually be more complicated to implement and modify, while also being less efficient in processing the algorithm. We present our concerns and justify them in the following sections.

6.2.1 Multiple Assignments to a variable.

As discussed in Chapter 5, problems exist when a program performs multiple assignments to a single variable. Using the SSA form complicates these problems. During the CFG to SSA conversion process, the result of each assignment is converted to an SSA name. Since the SSA name for each assignment is unique, optimizations may perform transformations that are not allowed. Consider the following graphs.

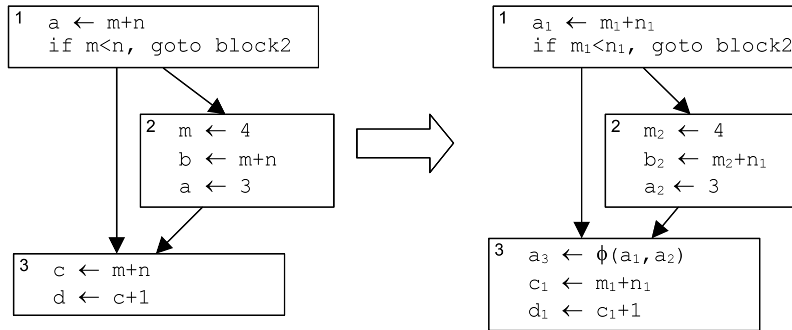


Figure 6-3. A control flow graph with multiple assignments to a variable.

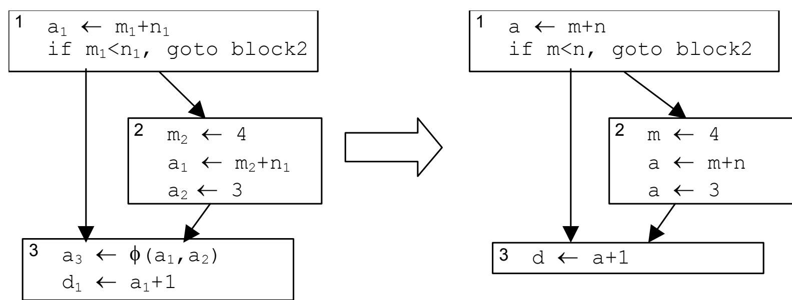


Figure 6-4. Partial redundancy incorrectly applied to figure 6-3.

Figure 6-4 shows how lazy code motion might optimize the SSA graph of figure 6-3. This approach produces incorrect code. Two solutions to this problem are the introduction of phi nodes and maps from SSA names to original CFG names. Each has its advantages and disadvantages.

Figure 6-5 shows how the SSA graph could be transformed correctly by adding phi nodes. This transformation has the advantage of being slightly more effective in certain circumstances. However, the inserted PHI instructions are more likely to cause the SSA to CFG conversion process to generate extra move instructions. Figure 6-6 continues the above example by showing how it might be transformed back into a control flow graph. Notice the added move instruction in block four. The move instruction is necessary to convert the phi instruction added to the SSA graph.

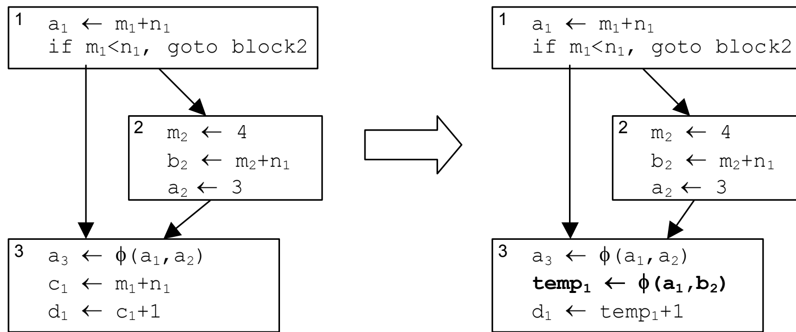


Figure 6-5. Insertion of PHI instructions during partial redundancy elimination

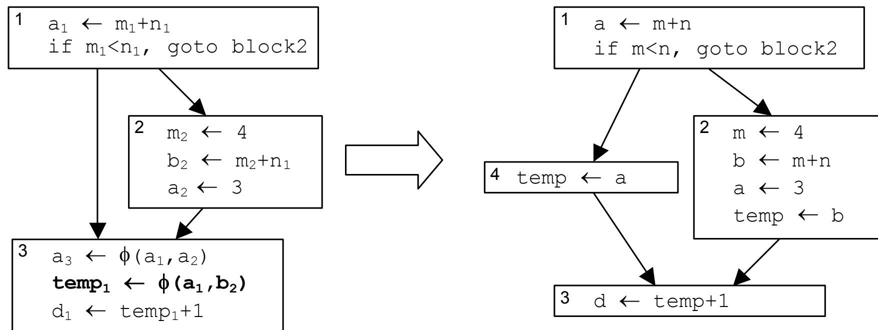


Figure 6-6. Proper transformation of Figure 6-3 to eliminate partial redundancies

If SSA to CFG conversion is done often, a large number of these transformations will introduce a large number of unnecessary moves. While it is possible for register allocation to eliminate many of the added moves, many remain. By applying the SSA to CFG conversion process repeatedly, which we will see that we must do, these added move instructions become more of a detriment to effective optimization.

A much simpler and no less effective solution is to maintain a map of SSA names to original variable names. A count can then be taken of how many SSA names map to an original variable name. From this count, we can deduce which SSA names map to variables that are assigned multiple times and act accordingly. This approach matches the lazy code motion algorithm in Chapter 5 and will produce the same results. Its drawbacks are that it requires access to such a map of SSA names, and it requires data flow analysis similar to that of non-SSA lazy code motion. With this approach, we can no longer take advantage of the dataflow analysis provided by the SSA graph.

6.2.2 Volatile, Global, and Referenced Variables in SSA form.

Earlier, we discussed adding the ability to perform optimizations on expressions containing volatile, global, and referenced variables (section 5.6). Performing optimizations with these variables on a SSA graph presents more difficulties.

Certain CFG to SSA conversion utilities may replace volatile, global, and referenced variables with SSA names. These SSA names will likely not have the properties that the

original variable names had. An algorithm must then demand access to a SSA name map such as that discussed in section 6.2.1 to retrieve any special properties of variables.

In our SSA libraries, volatile, global, and referenced variable names are not replaced with SSA names. This in effect creates invalid SSA graphs for any CFG that contains volatile, global or referenced variables. Figure 6-7 shows a CFG graph and a SSA representation of it. Notice that variable 'a' is assigned twice.

```
volatile int a = init();
int b = init();
if( a > b ){
    a = 5;
    b = 5;
}
```

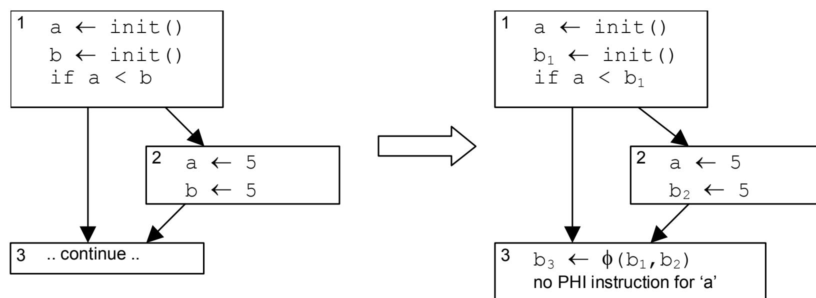


Figure 6-7. An invalid SSA graph generated due to a volatile variable.

In either approach to handling the SSA graph, dataflow analysis must be performed as if the graph is not in SSA form. For our lazy code motion algorithm, all the work that is involved to handle volatile, global, and referenced variables in a regular CFG is also required for handling these variables in the SSA graph and all of the advantages the SSA form offers are lost.

6.2.3 Rebuilding the SSA CFG.

Another problem with developing optimization algorithms involves the need to rebuild the SSA graph. Rebuilding the graph is a complex process having at least $O(n^2)$ complexity. Ideally, converting from CFG to SSA and back again is something necessary only once during compilation. We have found that this conversion to be necessary between optimization passes. It can also be necessary within an optimization algorithm if that algorithm requires several passes through a program's code. This is the case for our partial redundancy algorithm.

The difficulty lies in the need to rebuild an invalidated SSA representation. Since SSA requires that a certain value only be assigned once, any changes to the SSA graph must be made so that this property remains true. Maintaining this property involves inserting, removing, and modifying phi nodes. Inserting and removing phi nodes requires updates to the instructions that use the results of those phi nodes. All these modifications require complex data flow analysis, which require fairly complicated algorithms and lots of processing time.

Here, we present the problem of using SSA with the lazy code motion algorithm. Figure 6-8 shows a sample control flow graph and its corresponding SSA CFG.

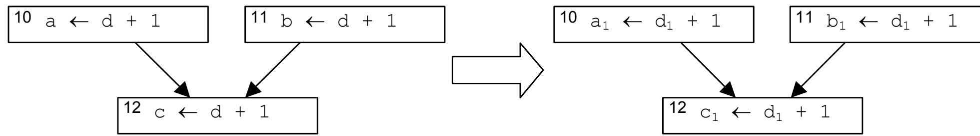


Figure 6-8. A sample control flow graph with no memory access expressions and its SSA equivalent

In Figure 6-8, the instructions in each block evaluate to the same expression. Partial redundancy would dictate that these expressions should then assign to the same variable. We'll assume the algorithm chooses a_1. After the lazy code motion pass is complete, the program code would transform into Figure 6-9. A_1 is now being assigned twice which makes the SSA graph invalid.

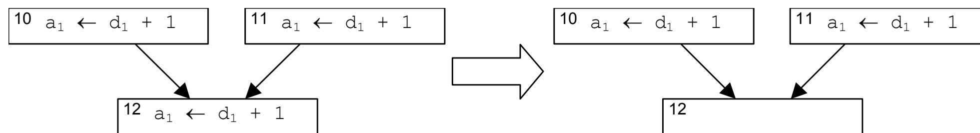


Figure 6-9. The invalid SSA graph in Figure 6-8 generated after expression naming and redundancy elimination

To correctly transform the SSA graph, a phi node must be inserted in block 12 to merge the results of block 10 and block 11, as shown in Figure 6-10.

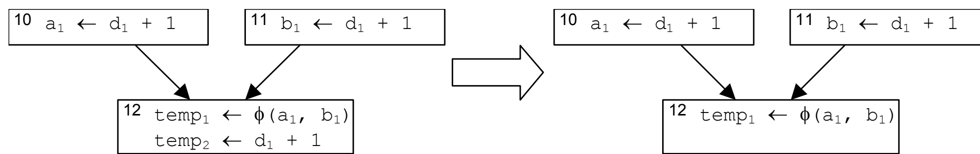


Figure 6-10. PHI instruction added by partial redundancy elimination to maintain SSA validity

While an algorithm for performing these kinds of transformations is possible, it turns out to be no more complex and time consuming to allow partial redundancy to invalidate the SSA graph (as in Figure 6-9), transform the intermediate code into CFG form and transform it back in to SSA form for the next partial redundancy elimination pass. Fortunately, our SSA libraries allow this approach. Other SSA libraries may not allow a SSA graph to be invalidated under any circumstances, making PHI instruction insertion mandatory.

6.3 SSA Conclusions

An optimization algorithm that uses the SSA form of a control flow graph proves to be, under certain conditions, more difficult to implement, with a greater complexity in processing time. These are the very properties that SSA research has intended to simplify. In our implementation of Lazy Code Motion, we find the conditions are not met for making SSA implementations worthwhile.

We have no detailed test results of efficiency between lazy code motion using an SSA graph and lazy code motion using a regular CFG. However, basic testing indicates that processing time of the algorithm using the SSA form averages 2-5 times that of the algorithm that operates on a standard control flow graph. Much of this performance degradation is due to conversion from and to SSA form on each pass of lazy code motion, which averages three times per procedure. As these conversions are exponentially complex, it is likely that larger functions will cause even worse performance of an SSA algorithm. The ability of the two algorithms to remove redundancies is identical.

With all these disadvantages to using the SSA form of a graph and as SSA provides no significant implementation advantages over a regular CFG, we use the original control flow graph for most optimizations. Still, the SSA form provides some advantages in developing algorithms. Since SSA is still an immature approach to writing compiler optimizations, there is a great deal of work and research yet to be done in developing SSA libraries and in use of those libraries. Therefore, we should not discount the usefulness of the SSA form based purely upon the findings of this Chapter.

Instead, we choose to carefully examine algorithms before choosing an SSA or CFG implementation. From our work with SSA and lazy code motion, we've developed some rules for determining if SSA is helpful in implementing an algorithm.

1. Will the algorithm require multiple passes over program code?

   If all work can be done in a single pass, the SSA graph will not need to be rebuilt, and the problems discussed earlier will not be pertinent.

2. How will the SSA graph transformations be made?

   If an algorithm must be applied to program code multiple times, changes to the SSA graph will need to be done so that the graph maintains a valid SSA form. Dead code elimination, for example, requires simple graph transformations. These transformations can be done so that the SSA graph remains valid.

3. Will the algorithm require special attention to certain variable characteristics?

   In our implementation of SSA lazy code motion, we have to handle any volatile, global, or referenced variables as if they are variables in a non-SSA graph. If these characteristics are not as relevant to an algorithm, or if input is known to not have these characteristics, an SSA approach may work well.

By considering these issues, we can determine whether there is any benefit to implementing an optimizing algorithm using the SSA form. For our example of lazy code motion, we find that the algorithm requires all of these conditions. Therefore, implementing lazy code motion may be simpler to implement on a non-SSA graph. A simpler optimization such as dead code elimination doesn’t impose as many restrictions, and therefore can easily be implemented on a SSA graph.

Chapter 7. Instruction Scheduling

Instruction scheduling is a technique that exploits the properties of a pipelined processor. The goal of an instruction scheduler is to arrange instructions so pipeline stalls can be avoided. The scheduler does not remove any instructions. It merely rearranges them. Of course, the scheduler can only rearrange instructions in a manner that preserves the results of program execution.

To understand the advantages of a code scheduler, consider the example of Figure 7-1. In this code, we have a memory read access. Memory accesses contribute significantly to pipeline delays in general-purpose processors, as modern processors have an order of magnitude difference between processor speed and memory speed. As our example instructions are ordered, the second instruction must wait for the duration of memory access delay before `r30` has the correct value. The pipeline must detect this dependency and pause execution until the result arrives. This is known as a pipeline stall.

```
ldw r30, 0(r10)
 -- stall waiting for memory access to complete --
add r30, r30, 1
```

Figure 7-1. An instruction sequence causing a pipeline stall.

Now consider the instruction sequence in the first column in Figure 7-2. Most of the instructions can be performed before, after, or during the read access. Assuming the read takes ten cycles to complete, nine instructions must be inserted between the read and any subsequent use of its result to avoid stalls in the pipeline. In the second column of Figure 7-2, we see one such organization of instructions that will avoid the pipeline stall. Notice that the read is moved earlier in execution order, while the write is moved later. In this particular example, all pipeline delays are avoided, reducing execution time from 21 cycles to 12.

```
cycle                            cycle
    1  ldc r5, 20                    1  ldw r30, 0(r10)
    2  ldc r10, _addr                2  ldc r5, 20
    3  add r10, r5                   3  ldc r10, _addr
    4  ldc r6, 4                     4  add r10, r5
    5  mul r10, r6                   5  ldc r6, 4
    6  ldw r30, 0(r10)               6  mul r10, r6
    -   - 9 cycle stall -            7  add r6, r6, 1
   16  add r30, r30, 1               8  add r5, r5, 1
   17  add r6, r6, 1                 9  ldc r7, 5
   18  add r5, r5, 1                10  mul r7, r7, r5
   19  ldc r7, 5                    11  add r30, r30, 1
   20  mul r7, r7, r5               12  add r6, r7, r5
   21  add r6, r7, r5
```

Figure 7-2. Reordering instructions to avoid pipeline a stall.

Many processors implement a caching scheme to avoid these stalls. Caching works well in the general case. However, programs with large data sets or volatile variables find that data cannot remain cached very long. A cache is only effective if data can be cached for extended periods of time. For better solutions, compiler intervention is required. Instruction scheduling is one such technique.

For a code scheduler to work well, it must have some understanding of the architecture with which it is working. A scheduler can do almost nothing with a processor that has no pipeline. If reads cannot be pipelined, then a scheduler should avoid issuing reads sequentially. A processor may have a separate pipeline for processing floating-point arithmetic. Pipeline stalls may not be predictable, forcing a scheduler to estimate stall times. The more complicated the architecture, the more optimization there is to exploit in code scheduling, and the more complex the scheduling algorithm must be.

We begin our design of a code scheduler by taking a closer look at the SCMP processor architecture. We will examine its pipeline properties in detail. Then we will consider those algorithms that have been developed for scheduling instructions and incorporate ideas into an SCMP code scheduler. Finally, we will examine our results and consider several changes to the architecture that might improve overall optimization potential.

7.1 The SCMP Pipeline.

In section 2.1.4, we introduced the SCMP architecture, including a brief discussion of the processor pipeline. Before we begin discussion of the SCMP code scheduler, we need more detailed information about this pipeline.

The SCMP processor uses an in-order four stage pipeline, as shown in Figure 7-3 [Rama1]. The pipeline stages are labeled instruction fetch (IF), instruction decode (ID), execute (EX), and write back (WB). The ALU, the portion of the processor that performs arithmetic and logic operations, operates between the ID and EX stages.

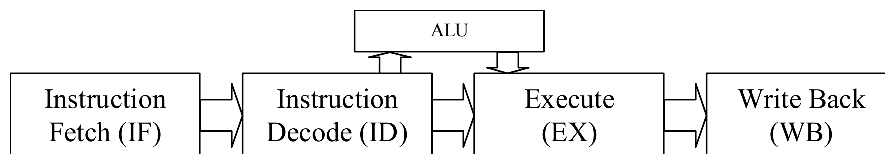


Figure 7-3. The SCMP pipeline

Assuming no stalls occur, the pipeline allows instructions to complete on every clock cycle. With a four-stage pipe, this means that as many as four instructions can be in progress at any given time. The pipeline manages register dependencies between these instructions. Register write after read, read after write, and output dependencies are not of any concern, as the pipeline will correctly forward values between stages. Figure 7-4 shows examples of these dependencies, with the dependent registers indicated. In each example, the SCMP pipeline can execute without any stalls.

```
add r5, r6, r6      add r5, r6, r7      add r5, r2, r3
sub r7, r5, r6      sub r6, r7, r7      sub r5, r4, r6
```

read after write (RAW)   write after read (WAR)   output dependencies

Figure 7-4. Examples of dependencies handled by the SCMP pipeline.

For no stalls to occur, an instruction must complete processing within a given pipeline stage within one clock cycle. If an instruction must consume a pipeline stage for more than one clock cycle for any reason, the pipeline must stall waiting for the instruction to complete. Instructions further in the pipe may continue, but all following instructions must wait for the stalled pipeline stage.

In the SCMP architecture, there are several scenarios in which the pipeline can stall. A memory read cannot complete and return its result before the register write back stage. This causes a dependency stall only if the following instruction requires the result of the read. Multiplies and divides cause pipeline stalls as they cannot be computed by the ALU within a single cycle. These are the instructions our code scheduler must consider while rearranging instructions.

Since each SCMP processor has local memory, access to that memory is fast. SCMP is designed to complete a load from or store to memory within two clock cycles. The access is passed to the memory controller in the decode state, which processes the access during the pipeline's execute stage. The execute stage is idle during this time. The result of a memory load becomes available during the write-back stage.

Since no registers are modified as the result of a memory write, it cannot cause a pipeline stall, though memory loads may. If the instruction in the ID stage requires the result of a memory access that is in the EX stage, a one-clock cycle delay is required so that the memory access can reach the WB stage and forward the result to the dependent instruction. Figure 7-5 demonstrates this delay.

```
cycle
    1  ldw r3, 0(r2)
    2   -----------    stall imposed while memory is read
    3  add r4, r3, 1
```

Figure 7-5. Pipeline stall on a memory read access.

We refer to this as a data dependent stall. The stall occurs due to an instruction dependency on the memory read. In the current SCMP architecture, this is the only scenario for a dependant stall. Fortunately, memory read instructions can be pipelined. A second memory access instruction (load or store) can be issued before the first has completed, so long as the second access does not depend on the returned value of the first. Figure 7-6 shows how memory accesses can be pipelined to avoid stalls.

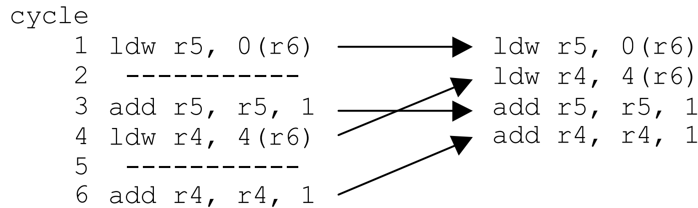


Figure 7-6. Reorganizing loads to avoid pipeline stalls.

Multiply and divide instructions consume a great deal of time on the SCMP processor. The SCMP arithmetic unit is estimated to take seven cycles to complete an integer multiply and twenty-one cycles to complete an integer divide. These delays cause a six or twenty cycle stall respectively for an integer multiply or divide. These stalls are not due to any data dependency. The stalls occur regardless of the sources or destinations of each instruction, and they occur regardless of the preceding and following instructions. Since these delays cannot be avoided, we refer to them as independent stalls. Currently, there is no advantage in the SCMP architecture to addressing independent stalls. However, changes to the architecture may make this information pertinent. We will attempt to design our scheduler to make use of them.

7.2 The Scheduler Algorithm

The SCMP scheduler contains two parts: a dependency tree builder and a scheduler. The dependency tree creates a partial order of instructions based upon dependencies between them. The tree allows us to determine how we may rearrange instructions while maintaining a valid instruction order. The scheduler then selects instructions from the dependency tree in a manner than preserves the tree's partial ordering and minimizes pipeline stalls.

Before designing the scheduling algorithm, we will make several assumptions:

- We will design our instruction scheduler to be applied after register allocation. Scheduling prior to register allocation leads to additional complications. We discuss some of the benefits and complications of scheduling prior to register allocation in section 7.3.1.

- Our scheduler will not attempt to move instructions between basic blocks. Algorithms that move instructions in this manner have the potential for worsening program performance. As the quantity of dependent stalls in the SCMP architecture are rare, we have little to gain by being able to move instructions between blocks. Our scheduler will move instructions within each basic block individually.

- We will also assume that, at most, there is a single branch located at the end of each basic block. Our algorithm can be easily adapted to handle a single branch at the beginning of the basic block instead of at the end. We assume that at most, there is one branch in the block. Branching instructions include all test-and-branch instructions, subroutine calls and returns.

7.2.1 Determining Possible Orderings

We begin our scheduler by establishing how instructions may be reordered. Valid orderings are dependent on memory accesses, register usage, and other dependencies between instructions. To obtain the most efficient schedule, we need an effective method for representing all possible valid orderings of instructions. By using a partial ordering of instructions to represent dependencies, we can efficiently determine what order of instructions is valid.

A partial order of any set determines dependencies between certain elements. Those elements of the set that have no partial order established between them can be reversed while maintaining the partial ordering. Examples of a partial ordering and several valid orderings derived from it are shown in Figure 7-7.

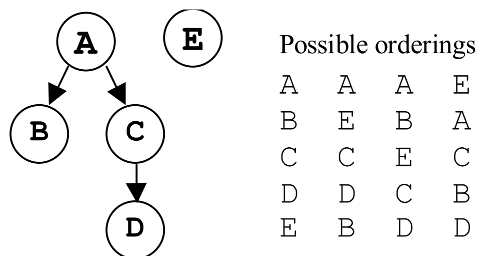


Figure 7-7. Several orderings resulting from a partial ordering.

We represent our partial ordering in the form of a graph, with nodes representing instructions and edges representing dependencies between instructions. Given any two nodes with an edge between them, we cannot reverse the order of those instructions. The edges of our graph are directed to maintain the original ordering.

To assign edges between the nodes of our dependency tree, we must determine all dependencies between instructions. We divide the discussion of our dependency detection into two groups: register dependencies and memory access dependencies.

Register Dependencies.

A register dependency is created when a register is assigned by one instruction and used or assigned by another. Examples of register dependencies are shown in Figure 7-4. These dependencies do not impose stalls on the pipeline. They merely limit the number of orderings we can derive by imposing dependency edges between instructions. The original order of two instructions cannot be reversed if

- the destination of first instruction is used as a source or destination in the second.

- the destination of the second instruction is used as a source or destination in the first.

Of course there is no need to indicate a dependency between a register read and a register write if between them there are numerous writes to the same register. To reduce

redundancies in our graph, we only maintain dependencies between a register access and the previous and successive register writes.

No dependencies exist between instructions that read the same register. Since a register read does not change the value of the register, successive reads may be ordered in any fashion. However, in generating our dependency tree, we must indicate each of the successive reads as being dependant on the previous write, while the next write to the register must be dependant on the previous write as well as each of the reads. We demonstrate this in Figure 7-8.

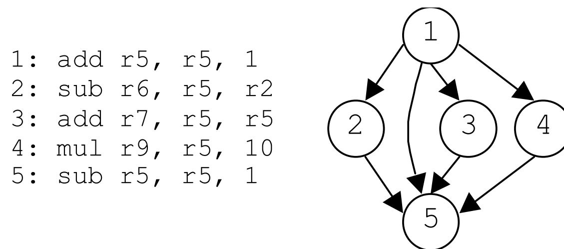


Figure 7-8. Successive reads must be dependent on previous and successive writes

For our partial ordering, we do not have to record how instructions are dependent. The dependency information we must record includes what dependencies exist as well as the registers creating each dependency. If two instructions have multiple dependencies, we must record each separately.

Memory Access Dependencies.

Having the ability to reorder memory access instructions is critical in our scheduler. We are attempting to avoid many of the stalls created by memory accesses, and that may involve rearranging memory accesses. Since we are performing our scheduling after register allocation, such high level programming concepts as local and global variables, volatile variables, arrays and data structures no longer exist. All instructions use registers or immediate values, making dependencies between memory accesses more easily determinable.

Dependencies between memory accesses are similar to dependencies between register accesses. As with register reads, memory loads are not dependent upon one another. The order of two loads can be reversed regardless of the memory being referenced. Two memory stores to the same address, or a memory load and a memory store to the same address cannot be reversed.

To determine whether two instructions access the same memory, we must attempt to determine the effective address of each memory reference at compile time. The address of every memory reference is determined by an effective address calculation. For the SCMP architecture, the only type of effective address calculation is the base displacement calculation. This involves adding a constant offset to a register value.

While being able to calculate values for effective addresses is the most accurate method for determining equality of effective addresses, it is rarely useful. The runtime values of registers are not often determinable at compile time. When the value of a register used in an address calculation is not known, we must determine the equality of effective addresses (or rather, possible lack of equality) by considering the registers used. These are the rules we must follow to determine effective address equivalency when the runtime value of a base register is unknown:

- If two effective address calculations use different base registers, and the value of either register is unknown, it is possible that the two effective address calculations may be equal. The write instructions must be considered dependent.

- If two effective address calculations use the same base register, but the value of this register is modified between the address calculations, the memory writes must be considered dependent. An example of this scenario is shown in Figure 7-9. Register r5 is modified between the two memory writes, and the value of r5 cannot be determined in the second write. Since we cannot evaluate the second address calculation, we must assume the writes are dependent.

```
1: ldc r5, _ptr2_
2: stw 0(r5), r6
3: ldc r5, _ptr1_
4: stw 4(r5), r6
```

Figure 7-9. The memory writes dependent due to modified unknown base register value.

- If two effective address calculations use the same base register and a different offset, and the base register is not modified between the calculations, the two effective addresses will compute to different values. The two writes are independent.

```
1: stw 0(r5), r6
2: stw 4(r5), r6
```

Figure 7-10. The two writes are independent regardless of the value of r5.

We must make consideration for half-word and byte accesses. In the figure below, we have several instructions that are dependent even though they have the same base register and different offsets.

```
1: stw 0(r5), r6     1: sth 2(r5), r6     1: stb 7(r5), r6
2: stb 1(r5), r6     2: stb 3(r5), r6     2: stw 4(r5), r6
```

Figure 7-11. Pairs of writes that show dependencies due to mixed access types.

SCMP memory accesses must be aligned on proper boundaries. Word accesses must be word aligned and half-word accesses must be half-word aligned. This prevents any two word accesses from overlapping with the same base and different offset. The following examples contain improperly aligned accesses, and so are illegal instructions that need not be considered by our scheduler.

```
1: stw 5(r5), r7      1: sth r7, 7(r3)      1: ldw r1, 0(r5)
2: stw 4(r5), r8      2: ldw r4, 4(r3)      2: stw r2, 3(r5)
```

Figure 7-12. Invalid memory accesses that need not be considered.

As with register dependencies many memory access dependencies are implied. Dependencies between instructions in a basic block are often implied by other dependencies within the block. To avoid redundant dependencies in our tree, we use the algorithm below.

```
base: the base register in the current effective address calculation
disp: the displacement of the current effective address calculation
active_base: the base register used in the previous store
curr_store[]: array mapping a displacement to the previous store using active_base
    and disp.
prev_store[]: curr_store prior to modification of active_base.
curr_load[][]: array mapping a base register and displacement to a list of reads of
    that effective address.
mem_size( ): returns the size (in bytes) of memory to be accessed by a memory
    access instruction.

If the access is a load
    If base = active_base
        If there exists an instruction in curr_store[disp]
            Add a dependency between this access and curr_store[disp].
        Else for each store in prev_store
                Add a dependency between this access and store.
    Else the base is not the active base.
        for each store in curr_store
            Add a dependency between this access and store.
    add this instruction to curr_load[base][disp].
    If mem_size(access) = 2
        add this instruction to curr_load[base][disp+1]
    Else if the mem_size(access) = 4
        add this instruction to curr_load[base][disp+1]
        add this instruction to curr_load[base][disp+2]
        add this instruction to curr_load[base][disp+3]

If the access is a store
    If base = active_base
        If there exists an instruction in curr_store[disp]
            Add a dependency between this access and curr_store[disp]
        Else for each store in prev_store
                Add a dependency between this access and store.
    Else for each store in curr_store
        Add the dependency between this access and store.
```

```
        Add store to prev_store
        Remove store from curr_store.
    Add this instruction to curr_store[displacement]
    If mem_size(access) = 2
        add this instruction to curr_store[disp+1]
    Else if the mem_size(access) = 4
        add this instruction to curr_store[disp+1]
        add this instruction to curr_store[disp+2]
        add this instruction to curr_store[disp+3]
    Active_base <= base
    For each base_reg in curr_load
        For each displacement in curr_load[base_reg]
            If base != base_reg or (base = base_reg and (disp/mem_size(access)
              = displacement/mem_size(access))
                For each instruction in curr_load[base_reg][displacement]
                    Add a dependency between the access and instruction
```

In this algorithm, dependencies based on modified base registers are not recorded. These dependencies will be implied by the register dependencies discussed earlier. In our example in Figure 7-9, the modification of the register r5 will cause dependencies between instructions 1 and 2, 2 and 3, and 3 and 4. This partial ordering will enforce the memory write dependency between instructions 2 and 4 without explicitly stating the write dependency.

Figure 7-13 demonstrates the various memory access dependencies in a dependency tree. Dependencies represented by a dashed line indicate memory access dependencies that are implied by register dependencies. Register dependencies are omitted.

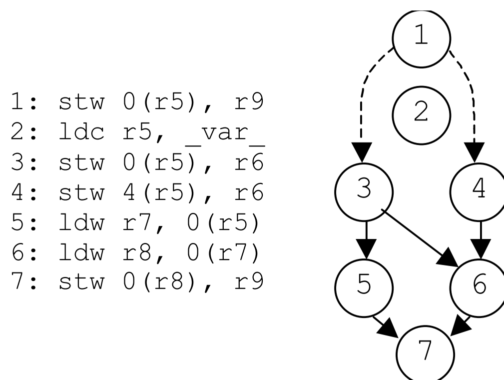


Figure 7-13. Several memory accesses and the resulting dependency tree.

```
 1: add r5, r6, r7
 2: sub r5, r5, 1
 3: stw 0(r3), r5
 4: ldw r4, 8(r2)
 5: mul r4, r4, 4
 6: add r7, r4, 21
 7: ldw r6, 9(r3)
 8: ldw r4, 12(r3)
 9: div r6, r6, r4
10: sub r3, r7, r6
11: stw 24(r2), r3
12: stw 32(r2), r7
13: mul r4, r4, r6
14: ldw r5, 28(r4)
15: add r5, r5, 1
16: ldw r6, 32(r4)
17: add r6, r6, 1
18: add r5, r5, r6
19: stw 0(r2), r5
20: ldw r7, 0(r3)
21: bne r7, _jump_
```

Figure 7-14. A sample program to be scheduled.

We present an example program in Figure 7-14. This segment of code is intended to include all possible dependencies that our dependency tree must represent. All previous examples addressed only portions of a tree. We provide Figure 7-15 as a complete dependency tree representing the program above.

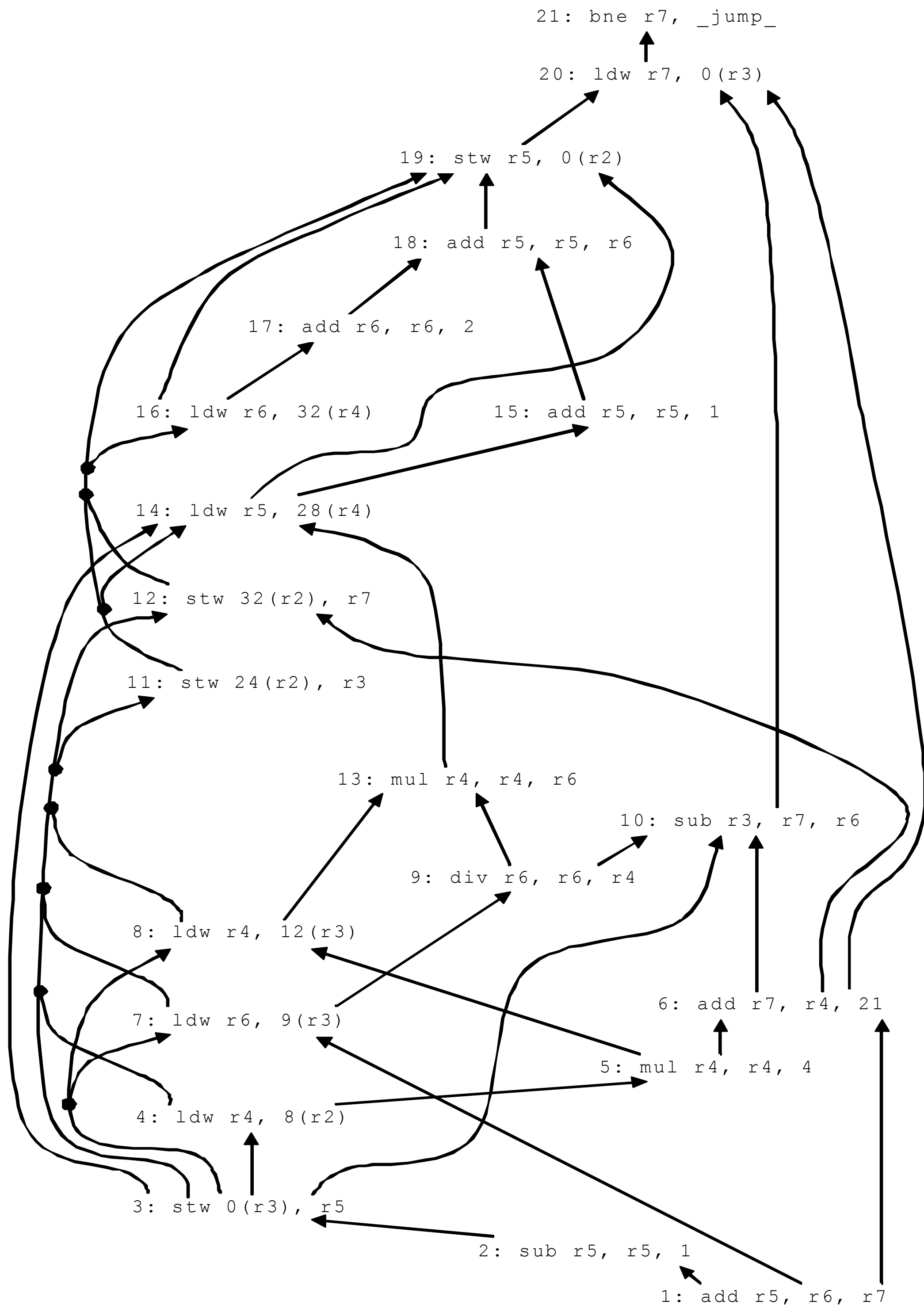


Figure 7-15. A Dependency tree of the program in Figure 7-14.

7.2.2 Scheduling Instructions

Once our dependency tree has been constructed, we must decide how to order the instructions to achieve a minimal number of pipeline stalls. To determine the optimal scheduling, we must have some idea of where pipeline stalls will occur given some ordering of instructions. The SCMP architecture is very predictable in this regard, as all stalls in an instruction order can be precisely determined at compile time. These stalls are a result of the multiple cycle instructions discussed in section 7.1.

We schedule instructions in reverse order, beginning with the branch instruction if one exists. As we schedule each instruction, we insert it before the previously scheduled instruction.

To select an instruction ordering, we divide the instructions in our dependency tree into three sets: placed, ready, and waiting instructions. A description of each set follows.

- **Scheduled Instructions.** As the name may suggest, scheduled instructions are those that have been assigned a position in the basic block. Since the last instructions of the block are scheduled first, the instruction's position is relative to the end of the block.

- **Ready Instructions.** Ready instructions have all their successors in the scheduled set, and are ready to be scheduled. All instructions begin in the waiting state, so it is sufficient to declare an instruction ready when its immediate successors (those successors represented in the dependency tree) are scheduled.

- **Waiting Instructions.** Waiting instructions have not had all successors scheduled. A waiting instruction can be moved to the ready set only when all its successors are scheduled.

The order of our new schedule is determined by the order in which instructions are removed from the ready set. An effective selection process is paramount in producing an optimal schedule. Selection will give priority to instructions based on the following criteria:

1.) Total dependency stall. This value refers to the number of stall cycles that can be avoided prior to this instruction. It is based upon the instructions that must be scheduled before it, and is independent of any stalls the instruction may cause. Instructions with the greatest total dependency stall have the most stall positions to be filled before them. Instructions with a greater total dependency stall are selected first to increase the possibility of other instructions filling the dependency stalls.

    An instruction's total dependency stall is difficult to determine during scheduling. We compute the total dependency stall when the dependency tree is generated and add this information to the incoming edges of the node. Then, for a given instruction, its total dependency stall time is the largest of the total dependency stalls of its incoming edges.

2.) Dependent stall time. Dependency stall time refers to the number of stall cycles an instruction imposes in a schedule. As these are dependency stalls, they are dependent on the instructions that have already been scheduled. Once an instruction is selected, this value represents the number of stall cycles that are introduced into the schedule. We select instructions with minimal dependency stalls to minimize the stalls introduced, and to allow the instruction (and its stalls) to fill dependency stalls during subsequent selections.

    An instruction's dependent stall time depends on the instructions that have been scheduled. We cannot include this information in the dependency tree. Dependent stall times must be calculated between instructions during scheduling.

3.) Independent stall time. If an instruction with dependency stall positions is executed prior to an instruction with an independent stall, the pipeline may begin the resource stall before the dependency stall instruction is complete. Since the resource stall is unavoidable, it can be used to fill the dependency stall positions.

    An independent stall is a static value that is determined by the type of instruction being executed. Since each node in our dependency tree already contains the instruction to be scheduled, it also contains the node's independent stall time.

4.) Dependency Count. We wish to keep our ready set as large as possible, so that we have a greater selection of instructions when attempting to fill dependency stall positions. To do this, we want to schedule instructions first that have the greatest number of instructions waiting on them. This gives more waiting instructions a greater chance entering the ready set. The total predecessor count gives us this value.

    As with the dependency stall time, an instruction's dependency count is easily determined when building the dependency tree. The set of instructions upon which an instruction is dependent is the union of the instructions upon which each processor is dependent in addition to each predecessor.

We distinguish the dependency count from the total dependency stall time. An instruction that is dependent on a large number of single cycle instructions will have a large dependency count, but its total dependency stall time will be zero.

We begin determining the order in which instructions are selected by establishing an order to the instructions in the ready set. By ordering these instructions, we avoid having to check all instructions when making a selection. The order gives priority to larger total dependency stall time first, and then smaller independent stall time and dependency count. Figure 7-16 shows an example of several instructions and how they would be ordered in the ready set.

	tds	is	dc
add r2, r2, 1	5	0	22
ldw r15, 4(r31)	5	0	20
mul r3, r4, r5	3	9	41
div r7, r9, r11	3	19	3
add r6, r4, r5	2	0	17
ldw r10, 0(r31)	2	0	5

tds – total dependency stall time
is – independent stall time
dc – dependency count

Figure 7-16. A list of ordered instructions in the ready set.

Since we are scheduling instructions in reverse order, the branch instruction must be scheduled first, if one exists. All additional instructions can be scheduled as follows:

```
While the ready set and the waiting set are not empty
    Select an instruction to move from the ready set to the scheduled set.
        For each predecessor of the selected instruction
            If the predecessor has all its successors in the scheduled set, move
              the predecessor to the ready set.
```

Our primary concern when scheduling instructions is to avoid pipeline stalls.

```
best_instr : instruction currently considered as the best selection
min_stall_time : stall created if best_instr is selected

for each curr_instr in the ready set
    stall_time = maximum stall if curr_instr is selected
    if( stall_time < min_stall_time )
        min_stall_time = stall_time
        best_instr = curr_instr
    if( stall_time == 0 )
        best instruction identified.  Stop search
```

In our algorithm, we stop searching for an instruction once we find one that avoids a pipeline stall at the current position in the schedule. We do not need to search further by virtue of the ordering of our ready set. Once an instruction is identified as causing no stall in the pipeline, we may stop searching the ready set. We know that any following instructions that also avoid stalls must have at most an equal total dependency stall and equal dependency count.

This reasoning follows in the event that we cannot find a ready instruction that avoids a pipeline stall. For whatever stall time that is the lowest possible among ready instructions, the first instruction in the ready set having that stall time is the best selection for the schedule. Subsequent instructions also having the same stall time must have at most the same total dependency stall and dependency count. However, we must continue searching the ready set for an instruction with a lower stall time, as that is our most important criteria.

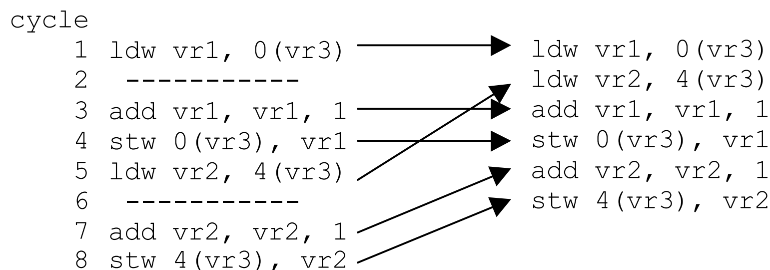
7.3 Scheduler Limitations

There are a number of limitations to the current SCMP instruction scheduler. These are problems not yet overcome in the algorithm that can potentially improve the algorithm's performance.

7.3.1 Scheduling before Register Allocation

The code scheduler presented in this chapter is designed for scheduling instructions after register allocation has been applied. While this simplifies the algorithm greatly, it severely limits the amount of optimization that can be performed. Figure 7-17 reveals how register allocation can limit the amount of potential code scheduling

Code scheduling before register allocation.



Code scheduling after register allocation.

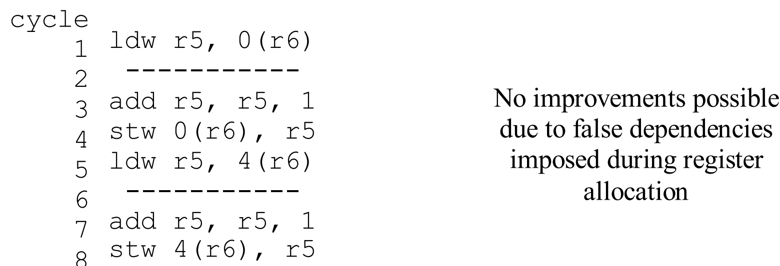


Figure 7-17. Affects of register allocation on code scheduling.

Unfortunately, to perform code scheduling before register allocation, the scheduler must consider register pressure when moving instructions. If an instruction is moved too far from the use of its variables, the register allocator must spill the register value. Not only does this eliminate the dependency the scheduler attempted to avoid, but it may also introduce a dependency and a stall that did not exist during code scheduling.

cycle	original code	after scheduling	after register allocation
1	ldw vr1, 0(vr10)	ldw vr1, 0(vr10)	ldw r1, 0(r0)
2	add vr1, vr1, 1	ldw vr2, 4(vr10)	ldw r2, 4(r0)
3	ldw vr2, 4(vr10)	ldw vr3, 8(vr10)	ldw r3, 8(r0)
4	add vr2, vr2, 1	ldw vr4, 12(vr10)	ldw r4, 12(r0)
5	ldw vr3, 8(vr10)	ldw vr5, 16(vr10)	ldw r5, 16(r0)
6	add vr3, vr3, 1	ldw vr6, 20(vr10)	------------
7	ldw vr4, 12(vr10)	add vr1, vr1, 1	**stw 24(r0), r5**
8	add vr4, vr4, 1	add vr2, vr2, 1	ldw r5, 20(r0)
9	ldw vr5, 16(vr10)	add vr3, vr3, 1	add r1, r1, 1
10	add vr5, vr5, 1	add vr4, vr4, 1	add r2, r2, 1
11	ldw vr6, 20(vr10)	add vr5, vr5, 1	add r3, r3, 1
12	add vr6, vr6, 1	add vr6, vr6, 1	add r4, r4, 1
13			**stw 28(r0), r1**
14			**ldw r1, 24(r0)**
15			------------
16			add r1, r1, 1
17			add r5, r5, 1

Figure 7-18. Register allocator introducing stalls after scheduling. Stall are not indicated until after scheduling.

cycle	original code	after register allocation	after scheduling
1	ldw vr1, 0(vr10)	ldw r1, 0(r0)	ldw r1, 0(r0)
2	add vr1, vr1, 1	add r1, r1, 1	ldw r2, 4(r0)
3	ldw vr2, 4(vr10)	ldw r2, 4(r0)	add r1, r1, 1
4	add vr2, vr2, 1	add r2, r2, 1	add r2, r2, 1
5	ldw vr3, 8(vr10)	ldw r3, 8(r0)	ldw r3, 8(r0)
6	add vr3, vr3, 1	add r3, r3, 1	ldw r4, 12(r0)
7	ldw vr4, 12(vr10)	ldw r4, 12(r0)	ldw r5, 16(r0)
8	add vr4, vr4, 1	add r4, r4, 1	add r3, r3, 1
9	ldw vr5, 16(vr10)	ldw r5, 16(r0)	add r5, r5, 1
10	add vr5, vr5, 1	add r5, r5, 1	**stw 24(r0), r5**
11	ldw vr6, 20(vr10)	**stw 24(r0), r5**	ldw r5, 20(r0)
12	add vr6, vr6, 1	ldw r5, 20(r0)	add r4, r4, 1
13		add r5, r5, 1	add r5, r5, 1

Figure 7-19. Scheduler avoiding stalls after register allocation. Stalls are not indicated until after scheduling.

Figure 7-18 demonstrates how performing code scheduling before register allocation can produce worse results if register pressure is not considered during scheduling. Assuming that only seven registers are available in a system, the allocator must spill registers after loading them. Register spills and subsequent fills are indicated. Code scheduling can be performed again after register allocation to eliminate newly introduced stalls. However, the additional register spills still exist.

7.3.2 Variable Stall Times

Another disadvantage of the SCMP scheduler is that it relies upon constant stall times. The SCMP pipeline supports this assumption, as all stalls are of constant time. The only

possible variable length stall is attributed to network communication, which instead triggers a context switch. If design changes to the SCMP processor make variable length stalls possible, a scheduler must be able to consider average, maximum, and minimum stall times when ordering instructions. This requires an entirely different scheduler.

7.3.3 Volatile Memory Dependencies

There is some concern about reversing the order of volatile memory reads. Our compiler loses knowledge of variable types after register allocation. This may have implications if memory values change after being read.

Such is the case with input streams. When an input buffer is read, a value is returned and a new value becomes available. The behavior exists because the memory address represents a buffer maintained by external hardware. As the SCMP architecture does not support such devices, this is not a concern. SCMP memory can only be changed when directly written from a local thread or network data message.

While external hardware may not yet pose a problem with the SCMP instruction scheduler, multiple threads create a concern. Volatile memory is declared as such within our applications because multiple threads and messages may write and read the same memory locations. If a program contains several volatile memory loads within a basic block, together with instructions that may cause the thread to suspend, program behavior may change.

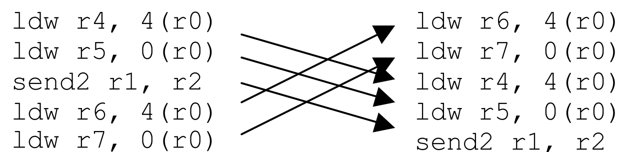


Figure 7-20. Movement of volatile loads may not be safe.

In Figure 7-20, the first column lists an original instruction order, while the second column represents a seemingly valid reordering. If the memory referenced by `r0` is volatile, and the send instruction causes the thread to suspend, another thread may modify the volatile memory. If this is the case, the code sequence in column 1 may not execute the same as the code sequence in column 2. While this is not a likely scenario, it is possible.

We leave this issue unresolved because we have no way of determining the types of memory being referenced after register allocation. Without specific type information, we cannot distinguish volatile memory accesses. We can prevent all memory reads from being reorganized, but this will severely inhibit the scheduler. Therefore, we will regard all memory reads as non-volatile reads.

7.4 Scheduler Conclusions

In this chapter, we have developed a basic instruction scheduler for the SCMP architecture. In developing the algorithm, we have discussed the details of the SCMP pipeline, the causes of pipeline stalls, and how we can go about removing those stalls to improve application performance.

In attempting to remove potential stalls, we have developed a method for reliably predicting where stalls must occur. In predicting those stalls, we have also shown that, by the nature of the SCMP pipeline, our scheduler will be very weak. Since memory loads are the only source of preventable stalls, our scheduler, and any other scheduler, will rely upon those loads to improve application performance.

In addition, we have proven that the MachSUIF infrastructure is insufficient for performing certain optimizations. MachSUIF does not retain enough information to correctly determine the use of variables, making it possible for correctly written algorithms to perform illegal code motion. That optimizations may turn valid code into invalid code renders the optimizations useless. Complications are entirely due to representations within MachSUIF, and can be alleviated by carrying necessary variable information through the entire compilation process.

Despite its difficulties and limitations, our the instruction scheduler can, even in its simplest form, provide a small but noticeable improvement to virtually all applications making it a worthy addition to the SCMP compiler.

Chapter 8. Testing and Results

We now present the results of the compiler with partial redundancy elimination and instruction scheduling.

8.1 Simulation Methodology

This section describes how SCMP simulations are performed and what the numbers in the following tables represent.

We provide three application kernels to obtain figures for our compiler performance: Fast Fourier Transform, Matrix Multiply, and Median Filtering. These kernels do not represent in themselves complete applications as might be executed on the SCMP processor. Rather, they provide data and computation intensive exercises that might be included in a complete application, thereby giving us simple estimations on how SCMP handles realistic tasks.

For each application, we simulate it with SCMP processors containing one, eight, sixteen, thirty-two, and sixty-four nodes. This provides some insight into how these optimizations affect SCMP communication. For SCMP simulations of one node, a sequential version of each application is executed, as multi-node synchronization can be avoided.

Execution times represent the number of simulation cycles required to execute each program. Each application records processing start and stop times (as determined by the simulator) and computes its execution as the difference of these values. This gives us simulation times that reflect delays due to network congestion, pipeline stalls, context switching, and all the major components of the SCMP processor.

The baseline compiler includes the SUIF and MachSUIF passes discussed in Chapter 3. Specifically, the baseline compiler includes optimizations for dead-code elimination and constant propagation. Our passes are applied in addition to the baseline compiler.

In Chapter 2, we discussed SCMP's lack of formal design concerning communication with external hardware. This imposes a serious limitation to testing, as we must provide some method for loading test data and storing results for validation. For our simulation purposes, we have created a method of communication based on the standard C input and output routines. SCMP test programs may using C routines for file reading and writing such as fprintf( ), getc( ), and scanf( ). Files are stored in the simulator's current directory. All nodes are allowed access to these files.

While our methods of simulated I/O do not reflect final design goals of the SCMP processor, they allow us to quickly and easily support various benchmarks written to use the standard C routines, and they speed program development. The primary limitations of SCMP (its on-chip memory and its network) are still imposed by the simulator. Since

the simulation times presented in this paper do not include time to load test files or to store their results, our simplified input and output techniques do not contribute to performance measurements.

8.2 Results

Table 8-1, Table 8-2, and Table 8-3 compare the simulation results of the baseline compiler, partial redundancy elimination, and instruction scheduling. Each table also provides the simulation time when partial redundancy elimination and instruction scheduling are both applied. All percentages are calculated as a difference from the baseline values.

	baseline	partial		scheduling		combined	
1	64,558,001	50,778,479	21.34%	63,476,657	1.67%	49,738,607	22.96%
8	8,179,646	6,425,554	21.44%	8,044,478	1.65%	6,295,513	23.03%
16	4,187,604	3,286,552	21.52%	4,120,020	1.61%	3,227,248	22.93%
32	2,140,293	1,671,225	21.92%	2,106,501	1.58%	1,638,438	23.45%
64	1,160,572	909,848	21.60%	1,143,677	1.46%	892,353	23.11%

Table 8-1. IFFT baseline simulation results.

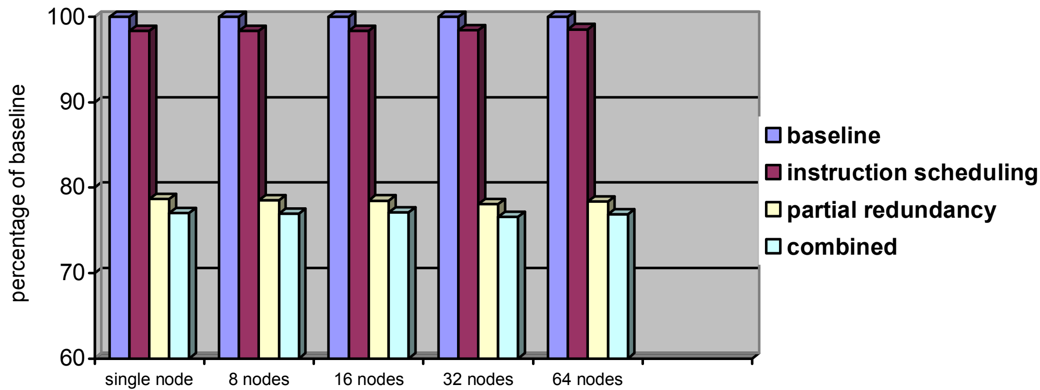


Figure 8-1. IFFT execution times as compared to baseline.

	baseline	partial		scheduling		combined	
1	18,920,543	17,889,245	5.45%	18,463,838	2.41%	17,497,564	7.52%
8	2,393,371	2,262,544	5.47%	2,336,022	2.40%	2,213,343	7.52%
16	1,209,683	1,143,791	5.45%	1,180,878	2.38%	1,119,070	7.49%
32	611,473	578,758	5.35%	596,940	2.38%	566,257	7.39%
64	312,842	296,218	5.31%	305,445	2.36%	289,837	7.35%

Table 8-2. Median Filter baseline simulation results.

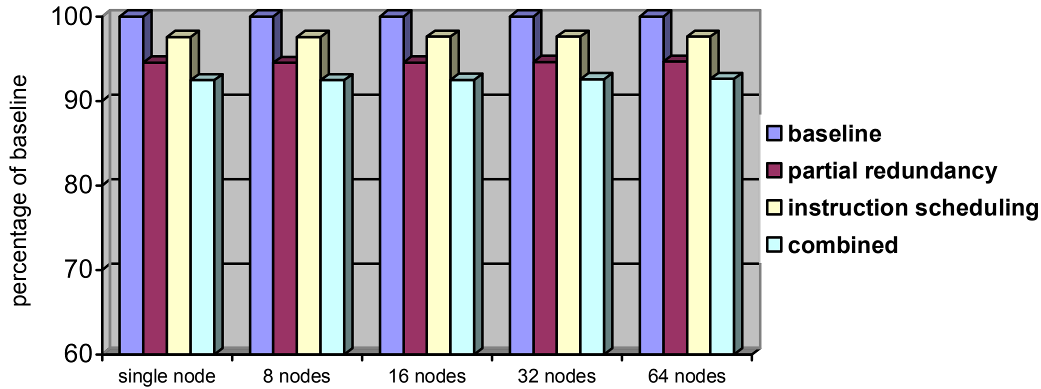


Figure 8-2. Median Filter execution times as compared to baseline.

	baseline	partial		scheduling		combined	
1	506,728,058	506,728,069	0.00%	489,688,185	3.36%	489,688,196	3.36%
8	65,670,584	65,677,452	-0.01%	63,415,584	3.43%	63,415,600	3.43%
16	32,522,537	32,517,592	0.02%	31,391,349	3.48%	31,386,645	3.49%
32	17,119,611	17,116,174	0.02%	16,513,488	3.54%	16,511,808	3.55%
64	8,402,866	8,402,646	0.00%	8,099,882	3.61%	8,099,096	3.62%

Table 8-3. Matrix Multiply baseline simulation results.

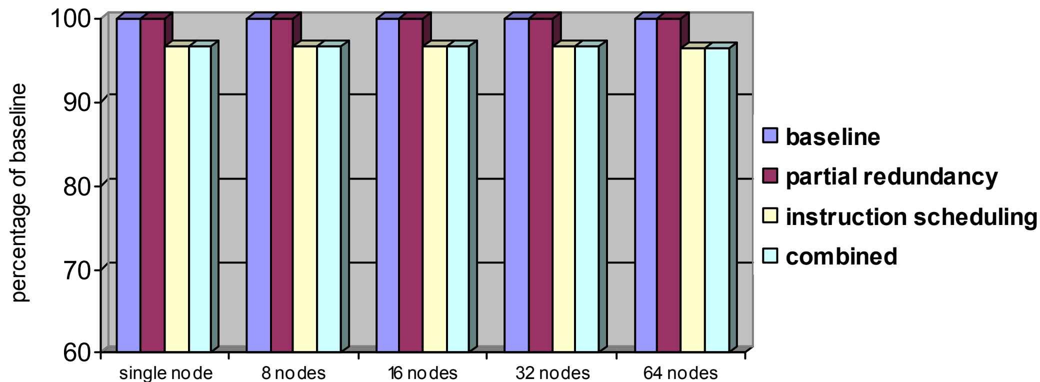


Figure 8-3. Matrix Multiply execution times as compared to baseline.

Immediately, several interesting properties of our optimizations emerge. IFFT reflects a modest improvement in performance when partial redundancy elimination is applied, while partial redundancy offers our matrix multiply kernel almost no improvement. As for instruction scheduling, improvements to matrix multiply are better than twice that of IFFT. In both categories, median filtering improvements fall between the other kernels.

The reasoning for partial redundancy's poor performances with matrix multiply is that matrix multiply uses a greater number of global variables. These cannot be removed without some degree of alias analysis. On a positive note, the extra loads generated from global memory access translate into dependent stalls that our scheduler can avoid.

The next three tables present test results of partial redundancy elimination with the unrestricted alias analysis as described in section 5.6. Results display the baseline compiler and the effects of partial redundancy with alias analysis. Combined results indicate partial redundancy elimination and instruction scheduling. Percentages reflect gains over partial redundancy without alias analysis and improvement over baseline values.

	baseline	partial w / aa			combined		
1	64,558,001	47,632,668	6.20%	26.22%	46,854,987	5.80%	27.42%
8	8,179,646	6,022,149	6.28%	26.38%	5,924,871	5.89%	27.57%
16	4,187,604	3,089,131	6.01%	26.23%	3,041,191	5.77%	27.38%
32	2,140,293	1,574,462	5.79%	26.44%	1,548,713	5.48%	27.64%
64	1,160,572	860,301	5.45%	25.87%	848,226	4.95%	26.91%

Table 8-4. IFFT simulation results including partial redundancy unrestricted alias analysis**.**

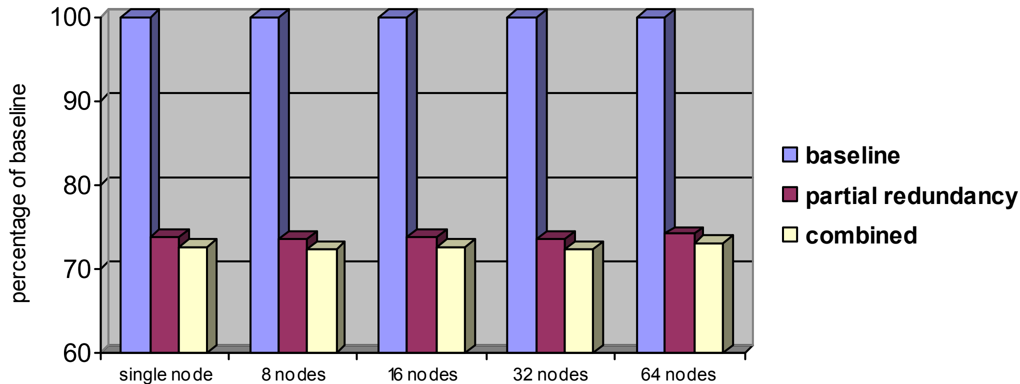


Figure 8-4. IFFT simulation results with alias analysis as compared to the baseline.

	baseline	partial w/ aa			combined		
1	18,920,543	16,691,442	6.70%	11.78%	16,040,689	8.33%	15.22%
8	2,393,371	2,111,416	6.68%	11.78%	2,029,831	8.29%	15.19%
16	1,209,683	1,065,954	6.81%	11.88%	1,025,041	8.40%	15.26%
32	611,473	539,833	6.73%	11.72%	519,236	8.30%	15.08%
64	312,842	275,667	6.94%	11.88%	265,238	8.49%	15.22%

Table 8-5. Median Filter simulation results including partial redundancy unrestricted alias analysis.

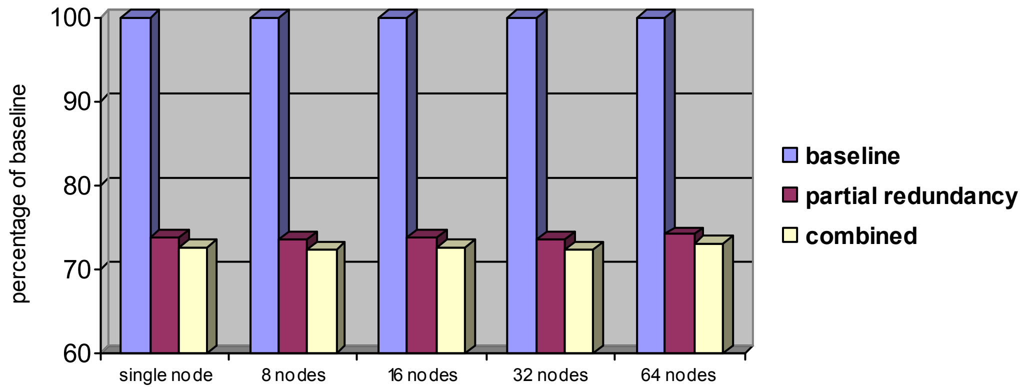


Figure 8-5. Median filtering results with alias analysis as compared to the baseline.

	baseline	partial w/ aa			combined		
1	506,728,058	338,305,915	33.24%	33.24%	321,396,602	34.37%	36.57%
8	65,670,584	44,364,179	32.45%	32.44%	42,165,019	33.51%	35.79%
16	32,522,537	21,872,950	32.74%	32.75%	20,775,352	33.81%	36.12%
32	17,119,611	11,718,521	31.54%	31.55%	11,144,062	32.51%	34.90%
64	8,402,866	5,706,130	32.09%	32.09%	5,417,421	33.11%	35.53%

Table 8-6. Matrix Multiply simulation results including partial redundancy unrestricted alias analysis.

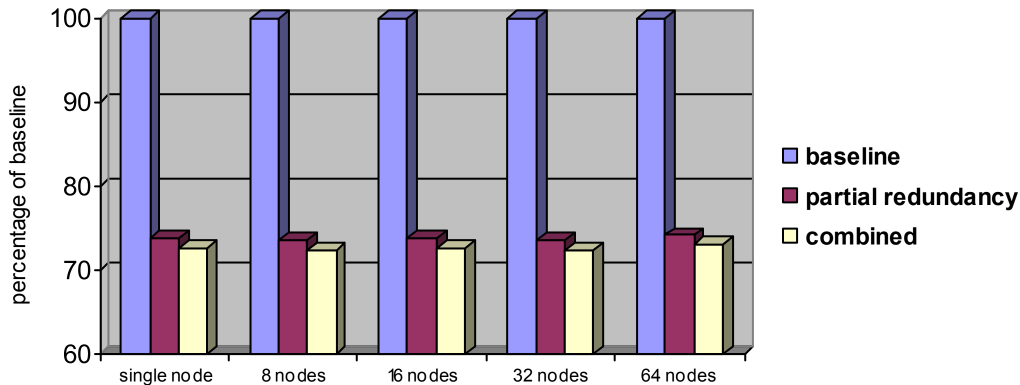


Figure 8-6. Matrix Multiply results with alias analysis as compared to the baseline.

With unrestricted alias analysis, we are able to remove many more redundancies from matrix multiply. Simulation times are reduced by more than thirty percent with partial redundancy elimination alone. Performance of partial redundancy elimination on our IFFT kernel improves marginally. A comparison of partial redundancy performance is shown on each application is shown in Figure 8-7.

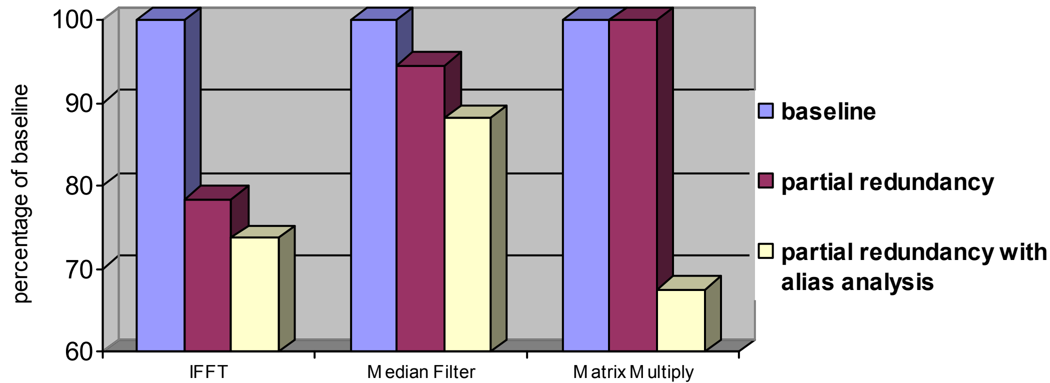


Figure 8-7. Aver age partial redundancy elimination performance of three application kernels.

8.3 Optimization Characteristics

In some simulations, we see a general decline in optimization effectiveness as the number of processors increases. In other tests, we see results that are worse than for simulations without our passes. These effects are not common to all of our simulations, and the effects are not significant (< 1%). However, some discussion of these occurrences is merited.

While it is possible for our partial redundancy elimination algorithm and instruction scheduler to produce code that is less efficient than baseline code, our optimizations are not directly the cause of the worsening performance. We have found under certain circumstances that by removing redundancies and performing additional optimizations, we can increase network congestion to the point that several messages are blocked for extended periods of time, resulting in longer execution times.

Determining the cause of this phenomenon could prove to be imperative, as we may find that we have a lower limit on optimization, beyond which further optimization becomes detrimental. Network design changes may also be necessary. At this time, no solution is known, as the cause cannot be determined in detail. The current simulator does not provide sufficient information to trace network messages. We must make improvements to network simulations before further testing can be done.

Chapter 9. Conclusions

9.1 Future Work

Of course, there is a great deal of research left to be done on the SCMP compiler. Here are several areas of focus that this thesis has mentioned but has left for future work. Some areas are discussed in more detail in their appropriate chapters.

9.1.1 Register Allocation

Research has been done on an interprocedural-based register allocation scheme [Chow88]. With this calling convention, the caller saves registers when necessary, freeing the callee from having to save registers the caller does not use. This calling convention has one major drawback: that a caller must know something of the callee's register. This sharing of information is difficult among widely used architectures such as Alpha, MIPS, or Intel. Since SCMP is designed to be part of an embedded or special purpose computer, it is likely to present a much more controlled programming environment. This makes the use of interprocedural analysis practical.

9.1.2 Instruction Scheduling

Our instruction scheduler must be applied after register allocation. Doing otherwise can severely worsen a programs performance given the current algorithm. Instruction schedulers that can take advantage of integrated register allocation and instruction scheduling will improve the effectiveness of removing pipeline stalls [Norris95] [Pinter93] [Good88].

A more robust scheduler will also allow us to examine the effects of pipeline modifications. As SCMP research continues, the structure of the pipeline is subject to change. Without the ability to consider these changes during instruction scheduling, we cannot fully realize the benefits of a modified pipeline. An effective scheduler is imperative when determining the value of additional pipeline hardware.

9.1.3 Additional Optimization

There are various optimization passes that prove to be very effective. These include algebraic simplification, distribution, and reassociation, as well as basic block reordering, interprocedural analysis, and compile-time branch prediction [Briggs94] [Hind99] [GNU]. Combining these optimizations with our existing partial redundancy elimination and instruction scheduling passes can decrease execution times to less than half that of the baseline. The SCMP compiler can benefit greatly by including implementations of these algorithms.

9.1.4 GCC Comparisons

In Chapter 3, we compared the advantages and disadvantages of the SUIF compiler infrastructure and the GNU Compiler Collection (GCC). GCC offers additional optimization passes and continuous development of sequential compilers. With a SCMP backend for the GCC compiler, we can take advantage of all that the GCC community offers. The backend for GCC would be very similar to the SCMP backend developed in this thesis. With the ability to compile programs with GCC for the SCMP architecture, we can draw a more concrete comparison between SUIF and GCC, possibly incorporating them into a larger compilation system.

9.1.5 Parallel Compiling

The primary goal of any compiler for a parallel processing architecture is to compile a sequentially designed application so that it can take advantage of the architecture's parallel nature. Such transformations require extensive data-flow analysis, as well as program and data partitioning. SUIF is designed to perform these transformations [Hall96]. As much research is being conducted in the area of partitioning for parallel processors, we can take much from the research community at large when developing a parallelizing SCMP compiler.

However, the SCMP architecture presents several unique conditions. We must consider the embedded multiprocessing, multithreading and message-passing features when compiling a parallel program for SCMP. As individual node performance improves, we also find that network congestion becomes an issue. Not only must we must consider these issues when performing parallel optimizations, but we must also make considerations for the SCMP architecture when making sequential optimizations such as partial redundancy elimination.

9.2 Summary

In this thesis, we have presented the design process and implementation of a compiler for the SCMP distributed-memory message-passing architecture. We have addressed the advantages and limitations of such architecture, and we have presented solutions for these concerns.

This thesis has presented the detailed design and implementation of the compiler backend for SCMP, as well as an implementation of partial redundancy elimination and a simple instruction scheduler for SCMP. We also discussed some limitations of using the Static Single Assignment form for implementing partial redundancy elimination that contradict some of the existing research literature. Finally, we presented results that showed performance improvements of 15-36% for a set of benchmark applications

Optimization passes must also consider the programming environment in which they are applied, and we have shown how to modify or design algorithms to consider the distributed memory architecture. As optimizations become more complex, networking

considerations become necessary. Giving the compiler additional strength in handling the multi-threaded synchronization proves valuable.

We can conclude that further exploration into the SCMP compiler is warranted. As hardware research continues, more extensive simulations will be required, giving rise to more complex programs. As these programs gain complexity, a compiler will need to be more capable of optimizing code. The above issue will also need to be addressed. SCMP architecture research has much to gain by a more robust compiler.

Bibliography

[ALPHA] "Assembly Language Programmer's Guide." Digital Equipment Corporation, Maynard, Massachusetts, March 1993.

[An1] S. Ananian, "Silicon C: A Hardware Backend for SUIF." Massachusetts Institute of Technology, May, 1998. http://www.flex-compiler.csail.mit.edu/SiliconC/

[Babb99] J. Babb et al., "Parallelizing applications into silicon." Field-Programmable Custom Computing Machines Proceedings, March 1999, pp. 70 – 80.

[Baker02] J. M. Baker et al., "SCMP: A Single-Chip Message Passing Parallel Computer." Proc. Parallel and Distributed Processing Techniques and Applications, PDPTA'02, CSREA Press, 2002, pp. 1485-1491.

[Briggs94] P. Briggs and K. Cooper, "Effective Partial Redundancy Elimination." ACM SIGPLAN Notices, vol. 29, issue 6, June 1994, pp. 159 –170.

[Briggs98] P.Brigss, T. Harvey, and L. Simpson, "Static Single Assignment Construction." John Wiley & Sons, Ltd., Jan. 1998. ftp://ftp.cs.rice.edu/public/compilers/ai/ssa.ps

[Chow88] F. C. Chow, "Minimizing register usage penalty at procedure calls" Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 85 – 94, 1988.

[Cytron89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck, "An efficient method of computing static single assignment form." ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 25 – 35, 1989.

[GCC02] R. Stallman, "GNU Compiler Collection Internals." Free Software Foundation, Boston, Massachusetts, Dec 2002.

[Gold03] B. Gold, "Balancing Performance, Area, and Power in an On-Chip Network", Master's Thesis, Virginia Tech, July 2003.

[Good88] J. Goodman and Wei-Hung Hsu, "Code Scheduling and Register Allocation in Large Basic Blocks." International Conference on Supercomputing, pp. 442 – 452, 1988.

[Hall96] M.W. Hall, et al., "Maximizing Multiprocessor Performance with the SUIF Compiler", IEEE Computer, vol. 29, issue 12, Dec. 1996. Also at http://suif.stanford.edu/papers/hall96.ps

[Hart02] A. Hartstein, T.R. Puzak, "The Optimum Pipeline Depth for a Microprocessor." Proceedings of the 29th Annual International Symposium on Computer Architecture, pp. 7 – 13, May 2002.

[Hind99] M. Hind, M. Burke, P. Carini, J. Choi, "Interprocedural pointer alias analysis", ACM Transactions on Programming Languages and Systems, vol. 21, issue 4, July 1999, pp. 848 – 894.

[Holl1] G. Holloway and M. Smith, "The Machine-SUIF SUIFvm Library." Division of Engineering and Applied Sciences, Harvard University, Oct. 2001.

[Holl2] G. Holloway, "The Machine-SUIF Static Single Assignment Library." Division of Engineering and Applied Sciences, Harvard University, July 2002.

[Kenn1] R. Kennedy et al., "Partial Redundancy Elimination in SSA form." ACM Transactions on Programming Languages and Systems, vol. 21, issue 3, May 1999, pp. 627 – 676.

[KRS1] J. Knoop, O. Rüthing, and B. Steffen, "Lazy Code Motion." ACM SIGPLAN Notices, vol. 27, issue 7, July 1992, pp. 224 – 234.

[Laio99] S.W. Laio, et al., "SUIF Explorer: An Interactive and Interprocedural Parallelizer" Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 37 – 48, 1999.

[Lam1] M. Lam et al., "An Overview of the SUIF2 Compiler Infrastructure." Computer Systems Laboratory, Stanford University

[Mitch99] N. Mitchell, L. Carter, J. Ferrante, and D. Tullsen, "ILP versus TLP on SMT." ACM/IEEE conference on Supercomputing, Article No. 37, 1999.

[Morel1] E. Morel and C. Revoise, "Global Optimization by Suppression of Partial Redundancies." Communications of the ACM , vol. 22, no. 2, Feb. 1979, pp. 96 – 103.

[Morgan98] R. Morgan, "Building an Optimizing Compiler." Morgan Kaufmann, 1997.

[Much97] S. Muchnick, "Advanced Compiler Design and Implementation." Morgan Kaufmann, 1997.

[Norris95] C. Norris, L.L. Pollock, "An experimental study of several cooperative register allocation and instruction scheduling strategies" Proceedings of the 28th Annual International Symposium on Microarchitecture, pp. 169 – 179, Nov. 1995.

[Patel00] S.J. Patel, "Increasing the Size of Atomic Instruction Blocks Using Control Flow Assertions." Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture, 2000, pp. 303 – 313.

[Pinter93] S. Pinter, "Register Allocation with Instruction Scheduling: a New Approach." ACM SIGPLAN Notices, vol. 28, issue 6, June 1993, pp. 248 - 257.

[Rama1] P. Ramachandran, "The SCMP Pipeline." Personal Communication, Virginia Tech, July 2003.

[Ras98] F. Rastello, A. Rao, S. Pande, "Optimal Task Scheduling To Minimize Inter-tile Latencies." Proceeding of the International Conference on Parallel Processing, pp. 172 – 179, Aug. 1998.

[SIA02] "The International Technology Roadmap for Semiconductors 2002 Update." Semiconductor Industry Association, 2002. http://public.itrs.net

[Smith1] M.D. Smith, G. Holloway, "An Introduction to Machine SUIF and Its Portable Libraries for Analysis and Optimization." Division of Engineering and Applied Sciences Harvard University, July 2002.

[Weg91] M. N. Wegman, F. K. Zadeck, "Constant propagation with conditional branches." ACM Transactions on Programming Languages and Systems, vol. 13, issue 2, April 1991, pp. 181 – 210.

Appendix A. The SUIFvm Instruction Set

Certain optimization passes presented in this paper use the SUIF virtual machine (SUIFvm) instruction set [Holl1]. In these chapters, the sample assembly code provided uses a simplified form of this instruction set. This appendix describes the format used in code the SUIFvm code samples.

Each instruction begins with the opcode, followed by a series of destination and source operands. In general, a source can be either variable name or immediate value. A destination must always be a variable name. Unless otherwise stated, the order of operands is as follows.

- If an instruction has a destination, the first operand is the destination, followed by the first source, the second source, and so on.

  ex.) add r5, r6, r7

- If an instruction has no destination, the first operand is the first source, followed by the second source, and so on.

  ex.) beq r5, r6, LABEL

Register names (r1, r2, etc) are often used in SUIFvm code samples. Since SUIFvm is not an instruction set for a real machine, there is no physical concept of registers. These register names can be regarded as compiler generated temporary variables.

LDA. Move the address of the source operand into the destination. The source can be a local or global variable name or a function name.

ex.) lda r1, global_variable_name

LDC. Move a constant value into the destination.

ex.) ldc r1, 8

LOD. Load the value at the address of the source into the destination. The source can include a variable containing an address and some constant offset in bytes.

ex.) lod r1, 8(r2)

STR. Store the source as the value at the address of the destination. The destination can include a variable containing an address and some constant offset in bytes.

ex.) str 10(r2), r1

MOV. Copies the value of the source to the destination.

ex.)   mov r2, r1

ADD. Adds the first and second sources and moves the result into the destination.

ex.)   add r1, r2, r3

SUB. Subtracts the second source from the first source and moves the result into the destination.

ex.)   sub r1, r2, 5

MUL, DIV. Multiplies or divides the first source by the second source and moves the result into the destination.

ex.)   mul r1, r2, r3

ex.)   div r1, r2, r3

MEMCPY. Copies the value from one memory location to another. The source and destination must be address expressions.

SEQ, SNE, SL, SLE. Compares the first and second source. The result of the comparison is stored to the destination. A non-zero result indicates true.

BEQ, BNE, BGE, BGT, BLE, BLT. Branch to target if the condition is true. The first and second sources are compared. If the result of the indicated comparison is true, execution continues at the indicated target label. If the result of the comparison is false, execution continues with the next instruction. The target label must follow the second source.

ex.)   bne r1, r2, NEXT_LABEL

JMP. Unconditional jump to the target label.

ex.)   jmp NEXT_LABEL

CALL. Calls the specified function with any listed sources. There is no need to setup a call stack in SUIFvm. Therefore, a call consists only of the function name followed by any number of sources in parentheses.

ex.)   call some_function(r4, r5, var)

RET. Returns from a procedure. If the function must return a value, that value must be the first and only source of the instruction.

ex.)   ret r1

Appendix B. SCMP RPC Library

These are the routines provided with the SCMP remote procedure call library.

parExecute(int num_threads, void (*fn)( ), … **)**

num_threads – number of threads to begin executing the remote procedure

fn – the remote procedure to be called.

varargs – additional arguments to be passed directly to each instance of <fn>.

parExecute( ) permits a programmer to call a remote procedure on multiple nodes. If <num_threads> represents the number of nodes on the system, one RPC is made to each node. If <num_threads> is greater than the number of nodes, RPCs are distributed as evenly as possible among all nodes. No consideration is made to the existing workload of the system.

All varargs are passed directly to the remote procedure. The programmer must take care to pass the arguments in the order that the procedure expects them.

parExecute( ) performs all synchronization internally. It does not return until all RPCs have signaled completion. This makes a return status is unnecessary.

createThread(int dst_node, void (*fn)( ), void (*callback_fn)( ), …)

dst_node – the node upon upon which the RPC will be made.

fn – the address of the remote procedure to be called.

callback_fn – the address of a procedure to be called remotely by <dst_node> to signal the RPC's completion. The call is made back to the node iniating the RPC. <dst_node> terminates immediately after starting the return RPC.

varargs – additional arguments to be passed directly to <fn>.

This routine assembles a SCMP thread message with the current node id, callback_fn, and remaining arguments and delivers the message to <dst_node>. Since this involves network communication, createThread( ) may cause the executing thread to stall if the message cannot be sent immediately. The routine returns after the thread message is sent. No return status is necessary, as the routine will wait until the thread message is complete.

All varargs are passed directly to the remote procedure. The programmer must take care to pass the arguments in the order that the procedure expects them.

This is for use when a programmer wishes to make a single RPC on a specific node. It can also be used when a programmer wants to implement a specific method for distributing multiple remote procedure calls.

endThread( )

This routine terminates the executing thread. The operation cannot fail and endThread( ) never returns. It can be used to prematurely end a thread. endThread( ) is only necessary when a thread must be explicitly stopped. The compiler inserts all necessary instructions to terminate a remote procedure.

Vita

Sidney Page Bennett was born on May 5, 1978 in Huddleston, Virginia. In 1996, he graduated from Staunton River High School in Moneta, Virginia. He continued his education at Virginia Western Community College, where he graduated Magna Cum Laude with an Associate of Science degree in General Engineering. He completed his undergraduate studies at Virginia Polytechnic Institute and State University with a Bachelor of Science in Computer Engineering, graduating Summa Cum Laude.

Sidney completes his Master of Science degree in Computer Engineering in November 2003, also at Virginia Tech, sponsored as a graduate research assistant under Dr. James Baker. He has committed to a year's service as a civilian employee of Naval Research Laboratories in Washington, D.C.