ENHANCING INSTRUCTION LEVEL PARALLELISM
THROUGH
COMPILER-CONTROLLED SPECULATION

BY

ROGER ALEXANDER BRINGMANN

B.S., University of South Alabama, 1983
M.S., University of Illinois at Urbana-Champaign, 1992

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1995

Urbana, Illinois

ENHANCING INSTRUCTION LEVEL PARALLELISM
THROUGH
COMPILER-CONTROLLED SPECULATION

Roger Alexander Bringmann, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1995
Wen-mei W. Hwu, Advisor

The available instruction level parallelism (ILP) is extremely limited within basic blocks of non-numeric programs [1] [2] [3]. An effective VLIW or superscalar processor must optimize and schedule instructions across basic block boundaries to achieve higher performance. An effective structure for ILP compilation is the *superblock* [4]. The formation and optimization of superblocks increase ILP available to the scheduler along important execution paths by systematically removing constraints due to the unimportant paths. Superblock scheduling is then applied to extract the available ILP and map it to the processor resources.

The major technique employed to achieve compact superblock schedules is speculative execution. Speculative execution refers to executing an instruction before knowing that its execution is required. Such an instruction will be referred to as a *speculative instruction*. In the general sense, speculative execution may be engineered at run-time using dynamic scheduling or at compile-time. Superblock techniques utilize compile-time engineered speculative execution, or speculative code motion. A compiler may utilize speculative code motion to achieve higher performance in three major ways. First, in regions of the program where insufficient ILP exists to fully utilize the processor resources, useful instructions may be executed. Second, instructions starting long dependence chains may be executed early to reduce the length of critical paths. Finally, long latency instructions may be initiated early to overlap their execution with useful computation. Speculative execution is generally employed by all aggressive scheduling techniques. For example, Tirumalai *et al.* showed that modulo scheduling of while loops depends on speculative support to achieve high performance [5]. Without speculative support, very little execution overlap between loop iterations is achieved.

This dissertation discusses the problems that must be addressed to perform compile-time speculation for acyclic global scheduling, classifies existing speculation models based upon how they solve these problems and discusses two new compile-time or compiler-controlled speculation models - write-back suppression speculation and safe speculation.

DEDICATION

To my wife, Mary, and my children, Carrie and Johnathan.

Thank you for your love and your support!

ACKNOWLEDGMENTS

First and foremost, I would like to acknowledge my advisor, Professor Wen-mei W. Hwu, for his intellectual, financial, professional, and emotional support. I value his advise and look forward to applying what I have learned from him as I continue my career in industry.

As a member of the IMPACT group, I was able to meet and work with a number of amazing people in industry. I would like to thank Mike Johnson and David Witt from AMD, David Ditzell and Steve Richardson from SUN MicroSystems, Jack Mills from Intel, and Mike Schlansker and Vinod Kathail for technical conversations and insights over the years. I would like to thank Dr. Bob Ramakrishna Rau for his valuable insights on instruction speculation and scheduling.

The research would not have been possible without the support of the IMPACT research group - past and present. I would like to acknowledge their support in the form of coding, practice talks, research discussions, and friends. Many thanks to Sadun Anik, Dave August, Pohua Chang, William Chen, Derek Cho, Tom Conte, Brian Dietrich, Dave Gallagher, John Gyllenhaal, Grant Haab, Rick Hank, Andrew Hsieh, Liang Hsu, Sabrina Hwu, Tokuzo Kiyohara, Dan Lavery, Scott Mahlke, Krishna Subramanian, and Yoji Yamada. I would also like to thank Vicki McDaniel, the secretary for the IMPACT group, for her friendship and assistance.

Next, I would like to acknowledge friends who have made graduate school an enjoyable experience. I would like to thank Neal Alewine for the friendship of a fellow southerner who was also going back to graduate school after a prolonged period of time. I would like to thank Scott Mahlke and Rick Hank for putting up with me as a room mate during my last year and a half. I would like to thank John Gyllenhaal and Liesl Little for the fun and pain provided by workout partners. Finally, I would like to acknowledge Sharad Mehrotra, Bill Pottenger and Krishna Subramanian for their friendship and support as we worked through the comprehensive exam process.

I would like to thank my parents, Norma and Wolfgang, for their encouragement towards excellence in school as well as their encouragement to pursue my doctoral dreams. I would also like to thank them for the emotional support when things were difficult. I would like to thank my brother, Michael, for the natural brotherly rivalry that helped motivate me to return for my doctorate after he completed his. I would also like to thank my in-laws, John and Elizabeth Cooper, for their support during graduate school.

Most importantly, I must thank my wife, Mary, and my children, Carrie and Johnathan, for their love. They stood behind me and believed in me when I decided to step away from a successful industry career and pursue my doctorate. They were there to help me enjoy the good times in my graduate studies and have provided the critical emotional support through the difficult times. The last year and a half have been very difficult for all of us and I hope that I can more than make up for this very soon!

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

Instruction scheduling is the process used by the compiler to re-order instructions in an effort to minimize program execution time. Since instruction scheduling is NP-Hard, heuristics are used to approximate the best schedule. One common approach to scheduling is to perform list scheduling using greedy heuristics to approximate a *globally optimal* schedule [6]. Regardless of the scheduling heuristics, instructions are ordered based upon some priority mechanism. At each cycle, the instructions with the highest priority that have resolved all dependences and meet the issue requirements of the processor are scheduled.

The implementation of a scheduler is straightforward if list scheduling is applied within basic blocks. Unfortunately, there is insufficient instruction level parallelism available within basic blocks of non-numeric benchmarks to fully utilize the functional units of wide issue superscalar and VLIW architectures [1, 2, 7]. Therefore global scheduling techniques such as *trace scheduling* [8] and *superblock scheduling* [4] have been proposed to permit greater scheduling and optimization freedom beyond basic block boundaries. Using these techniques, the program is divided into a set of traces or superblocks that represent frequently executed paths. These traces or superblocks contain multiple basic blocks and as a result can contain multiple conditional branches. When building a dependence graph for a trace or superblock, control dependence arcs are added from conditional branches to subsequent instructions. In order to gain additional scheduling freedom beyond the natural basic block boundaries found within these traces or superblocks, the compiler must remove some of these control dependence arcs. This permits speculation of instructions past conditional branches, thus the name compiler-controlled speculation.

When an instruction is speculated above a branch, it is executed regardless of the direction taken by the branch. To ensure correct program execution, the compiler must not alter the program execution result with speculative instruction scheduling. One requirement is that speculative instructions from one path of a conditional branch should not corrupt the source operands of instructions on the other path of the branch. This can be achieved by renaming the destination operands of the speculative instructions [4]. A more difficult requirement is that if the compiler speculates a potentially excepting instruction (PEI) from one path of a conditional branch, its exception condition must not alter the program state if the branch chooses the other path. In particular, the exception conditions from speculatively executed instructions must not corrupt the processor state or terminate the execution of the program unless their execution is confirmed by the subsequent branches.

This research focuses on compiler-controlled speculation models that have been applied to acyclic schedulers. The same speculation models provide the same benefits to cyclic scheduling models as well. The research topics were implemented in the IMPACT-I compiler developed at the University of Illinois [7]. The fundamental premise of the IMPACT project has been to provide a complete compiler implementation that allows quantification of the compiler and architectural extensions on the performance of superscalar and VLIW processors by compiling and executing large control-intensive programs.

1.1 Contributions

The four major contributions of this dissertation are discussed below.

- The problems that all compiler-controlled speculation models must solve are discussed. One of the major problems associated with all of these models is what should be done if an instruction that is beneficial to speculate can introduce run-time errors. For example, if a load is speculated above a branch that is preventing an illegal memory reference, the program could terminate during execution. These speculation models must either prevent speculation of these *potentially excepting instructions*, have some means of ignoring the exception until the direction of the branch is known, or ignore the exception. A classification for compiler-controlled speculation models is presented based upon the different approaches chosen to address this problem.

- Write-back suppression is an alternative to existing speculation models that require processor assistance to resolve exceptions resulting from compile-time speculation. This implementation takes advantage of in-order retire mechanisms present in many existing processors to systematically suppress register file updates for subsequent speculative instructions in the event of an error. This reduces the register pressure requirements which permits more aggressive speculation. Additionally, the technique requires much less processor support than existing techniques. Experimental results show that this technique can achieve performance results close to the most aggressive speculation models that ignore errors entirely.

- Safe speculation is presented as an alternative speculation model that permits aggressive compile-time speculation without processor extensions. This model uses intra-procedural and inter-procedural analysis to identify instructions that will not cause run-time errors and are therefore always safe to speculate. Additionally, the analysis methods provide information to later compilation phases to ensure that no transformations are performed that would invalidate the safety analysis. This permits aggressive speculation of memory references to dynamically allocated buffers within loops. This model can be used in conjunction with other speculation models such as write-back suppression to reduce the instruction overhead required to detect exceptions by identifying instructions that will definitely not cause an exception.

- Previous work on compiler-controlled speculation has ignored the possible cache effects introduced from speculation. Indeed, aggressive speculation can lead to significantly higher instruction and data cache misses. This work has shown that even though aggressive speculation does introduce higher cache miss rates, the overhead from these additional cache misses does not negate the benefits of using aggressive speculation models even with very small caches. A detailed study of the effectiveness of limited and aggressive speculation models is performed to show these effects for both instruction and data caches. These studies show the effects over a variety of processor issue rates and cache configurations.

1.2 Overview

This dissertation is composed of 7 chapters. Chapter 2 presents an overview of the organization and operation of the IMPACT Compiler. All compiler techniques discussed in this thesis are implemented within the framework of the IMPACT compiler.

Chapter 3 introduces the concept of compiler-controlled speculation. It describes the requirements that all compiler-controlled speculation models must address and classifies existing models based upon how they address these requirements. Examples speculation models are presented which fall into each of the speculation classes. Finally, an overview of the IMPACT scheduler is presented along with the global scheduling heuristics which take advantage of these speculation models. Experimental results are presented to demonstrate the effectiveness of these scheduling heuristics. Write-back suppression is presented in Chapter 4 to present an alternative compiler-controlled speculation model that requires some processor assistance to perform recovery.

Safe speculation is presented in Chapter 5 as an alternative speculation model that can additionally be used the enhance the performance of existing speculation models. This model takes advantage of intra-procedural and inter-procedural analysis to determine instructions that meet the requirements discussed in Chapter 3. The algorithms presented have been implemented and tested on a suite of C benchmarks. Additionally the effectiveness of these techniques are compared against a number of existing speculation models.

Chapter 6 studies how limited and aggressive speculation models may affect instruction and data caches. These studies are shown over a variety of block sizes, cache sizes and associativities. The results are analyzed and conclusions on these effects are presented. Finally, Chapter 7 presents conclusions and directions for future research.

CHAPTER 2

OVERVIEW OF THE IMPACT COMPILER

2.1 The IMPACT Compiler

A block diagram of the IMPACT compiler is presented in Figure 2.1. The compiler is divided into 3 distinct parts based on the level of intermediate representation (IR) used. The highest level IR, *Pcode*, is a parallel C code representation with loop constructs intact. The Pcode level is ideal memory dependence analysis [9], loop-level transformations [10], and memory system optimizations [11]. The middle IR is referred to as *Hcode*. Hcode is a flattened C representation with simple if-then-else and goto control flow constructs. In Hcode, statement level profiling is performed. Additionally, profile-guided code layout and function in-line expansion are performed at this level [12].

The final level of IR in the IMPACT compiler is referred to as *Lcode*. Lcode is a generalized register transfer language similar in structure to most load/store processor assembly instruction sets. Lcode is logically subdivided into 2 subcomponents, the machine independent IR, Lcode, and the machine specific IR, *Mcode*. The data structures for both Lcode and Mcode are identical. The difference is that Mcode is broken down such that there is a one to one mapping between Mcode instructions to the target machines assembly language. Therefore, to convert Lcode to Mcode, the code generator breaks up Lcode instructions into 1 or more instructions which directly map to the target architecture. The annotation of Lcode instructions to Mcode instructions is performed for a variety of reasons including limited addressing modes, translating

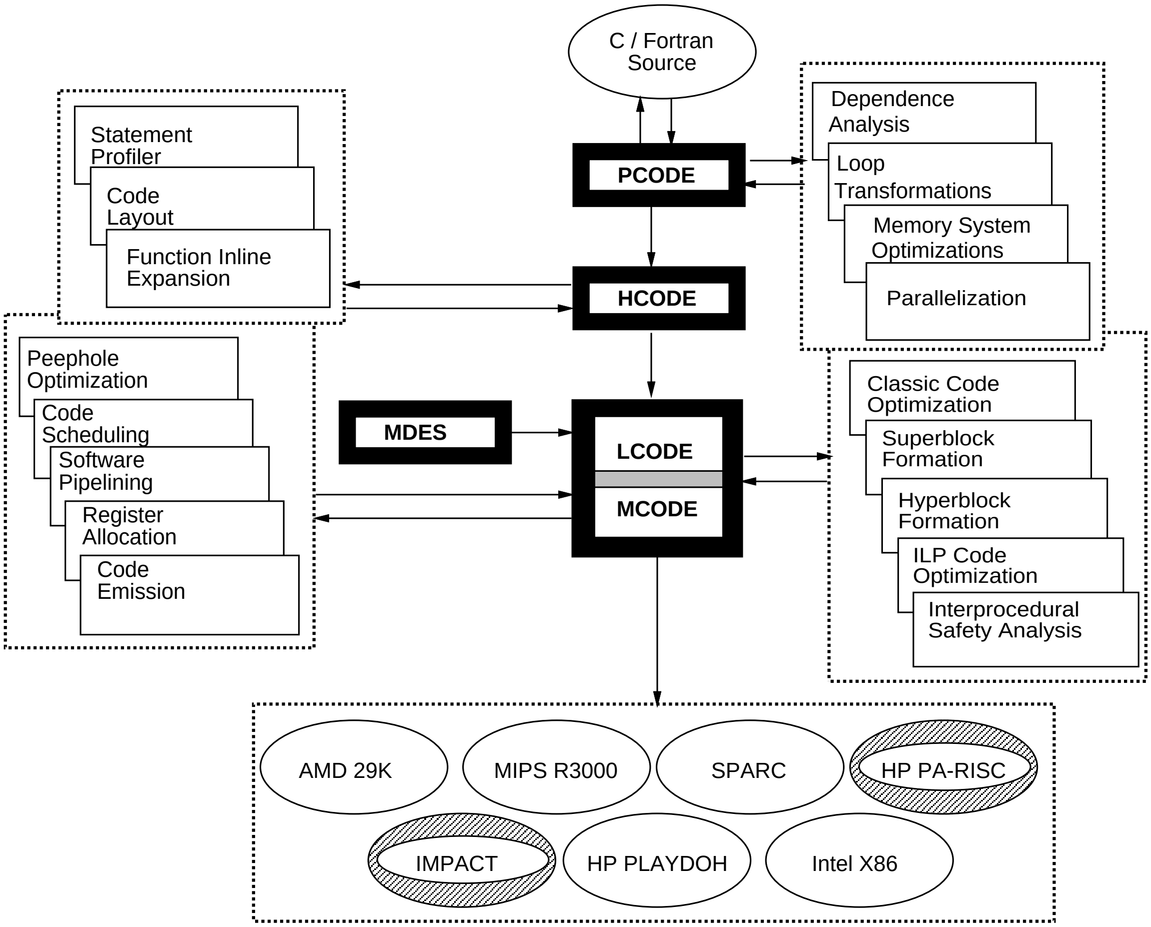


Figure 2.1 The IMPACT compiler.

unsupported Lcode instructions into their equivalent supported Mcode instructions, and literal size mismatch [13].

At the Lcode level, all machine independent classic optimizations are applied [14]. These include constant propagation, forward copy propagation, backward copy propagation, common subexpression elimination, redundant load elimination, strength reduction, constant folding, constant combining, operation folding, operation cancellation, dead code elimination, jump optimization, unreachable code elimination, loop invariant code removal, loop global variable migration, loop induction variable strength reduction, loop induction variable elimination, and loop induction variable re-association. Additionally at the Lcode level, the intra-procedural and inter-procedural safety analysis discussed in this dissertation are performed. This includes identifying safe instructions for speculation and function calls that do not modify memory (side-effect free).

Superblock and hyperblock compilation techniques are all performed at the Lcode level [15]. Superblock support includes superblock formation using execution profile information, superblock classical optimization, and superblock ILP optimization. When predicated execution support is available in the target architecture, hyperblocks rather than superblocks are used as the underlying compilation structure. All superblock optimization techniques have also been extended to operate on hyperblocks. In addition, a set of hyperblock-specific optimizations to further exploit predicated execution support are available.

All code generation in the IMPACT compiler is performed at the Lcode level. The two largest components of code generation are the instruction scheduler and register allocator. Scheduling is performed via either acyclic global scheduling [16, 17] or software pipelining using modulo scheduling [18]. For the acyclic global scheduling, code scheduling is applied both before register allocation (prepass scheduling) and after register allocation (postpass scheduling) to generate an efficient schedule. For software pipelining, loops targeted for pipelining are identified at the Pcode level and marked for pipelining. These loops are scheduled using software pipelining and all other code is scheduled using the global scheduler. Graph coloring based register allocation is utilized for all target architectures [19]. The register allocator employs execution profile information if it is available to make more intelligent decisions. For each target architecture, a set of specially tailored peephole optimizations are performed. These peephole optimizations are designed to remove inefficiencies during Lcode to Mcode conversion, take advantage of specialized opcodes available in the architecture, and to remove inefficient code inserted by the register allocator [13].

A detailed machine description database, *Mdes*, for the target architecture is also available to all Lcode compilation modules [20]. The Mdes contains a large set of information to assist with optimization, scheduling, register allocation, and code generation. Information such as the number and type of available function units, size and width of register files, instruction latencies, instruction input/output constraints, addressing modes, and pipeline constraints is provided by the Mdes. The Mdes is queried by the optimization phases to make intelligent decisions regarding the applicability of transformations. The scheduler and register allocator rely more heavily on the Mdes to generate efficient as well as correct code.

Seven architectures are actively supported by the IMPACT compiler. These include the AMD 29K [13], MIPS R3000 [21], SPARC [22], HP PA-RISC, and Intel X86. The other two supported architectures, IMPACT and HP Playdoh [23], are experimental ILP architectures. These architectures provide a framework for compiler and architecture research. The IMPACT architecture is parameterized superscalar processor with an extended version of the HP PA instruction set. Varying levels of support for speculative execution and predicated execution are available in the IMPACT architecture.

2.2 Superblocks

2.2.1 Superblock Formation

A superblock is a block of instructions in which control may only enter from the top but may leave at one or more exit points. When execution reaches a superblock, it is likely that all basic blocks in that superblock are executed. Superblocks allow the compiler to effectively optimize and schedule across basic block boundaries by removing the constraints due to side entrances within a sequence of basic blocks.

Superblocks are formed in two steps. Traces within a program (sets of basic blocks which tend to execute in sequence [8]) are first identified using execution profile information [24]. Tail duplication is then performed to eliminate any side entrances to the trace [25]. The basic blocks in a superblock need not be consecutive in the code. However, our implementation restructures the code so that all blocks in a superblock appear in consecutive order to the optimizer and scheduler.

Formation of superblocks is best illustrated with an example. Figure 2.2(a) shows a weighted flow graph which represents a loop code segment. The nodes correspond to basic blocks and arcs correspond to possible control transfers. The *count* of each basic block indicates the execution frequency of that basic block. In Figure 2.2(a), the *count* of $\{A, B, C, D, E, F\}$ is $\{100, 90, 10, 0, 90, 100\}$, respectively. The *count* of each control transfer indicates the frequency of invoking these control transfers. In Figure 2.2(a), the *count* of $\{A \rightarrow B, A \rightarrow C, B \rightarrow D, B \rightarrow E, C \rightarrow F, D \rightarrow F, E \rightarrow F, F \rightarrow A\}$ is $\{90, 10, 0, 90, 10, 0, 90, 99\}$, respectively. Clearly, the most

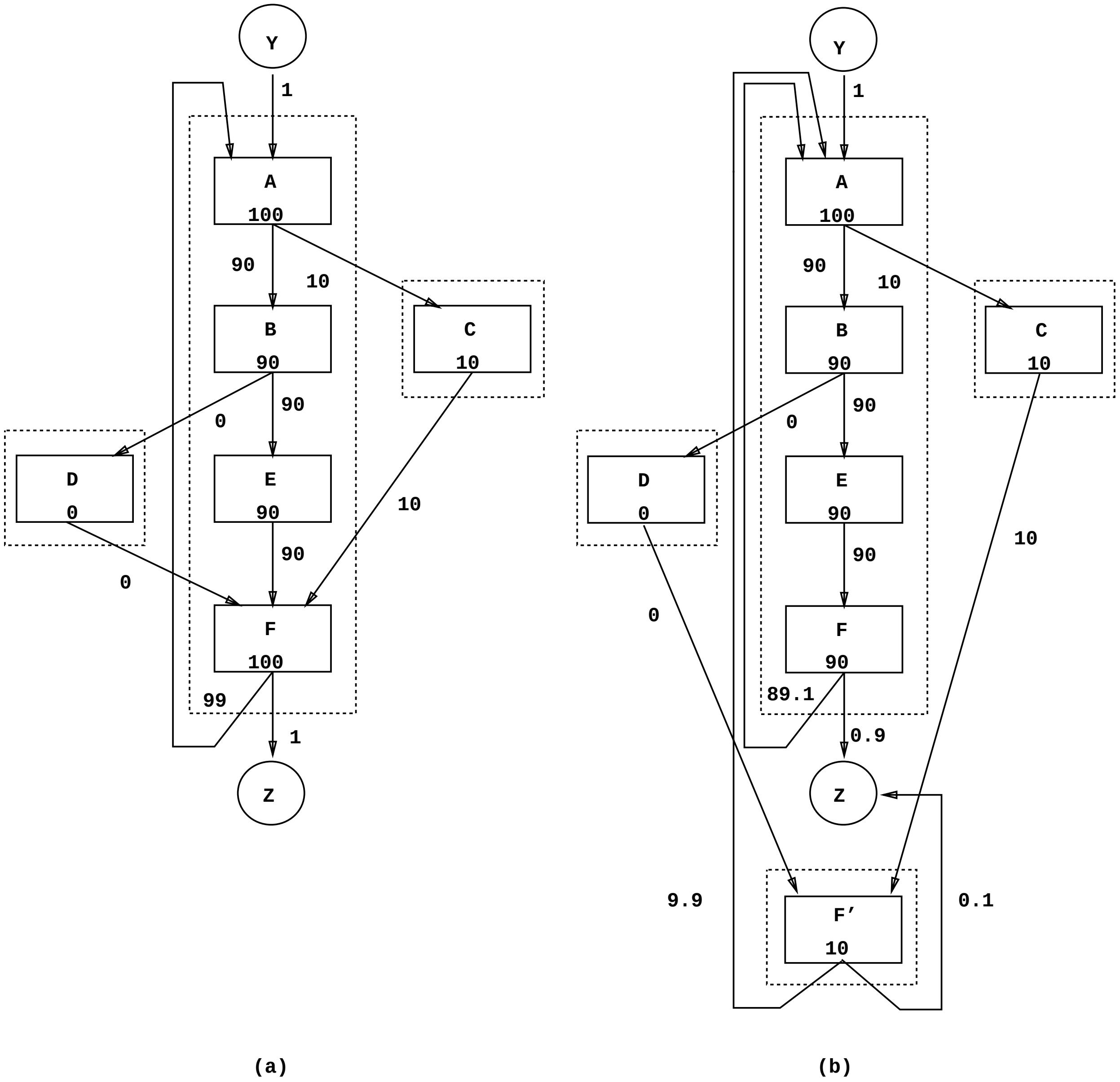


Figure 2.2 An example of superblock formation.

frequently executed path in this example is the basic block sequence $< A, B, E, F >$. There are three traces: $\{A, B, E, F\}$, $\{C\}$, and $\{D\}$. After trace selection, each trace is converted into a superblock by duplicating the tail part of the trace, in order to ensure that the program control can only enter at the top basic block. In Figure 2.2(a), we see that there are two control paths that enter the $\{A, B, E, F\}$ trace at basic block F. Therefore, we duplicate the tail part of the $\{A, B, E, F\}$ trace starting at basic block F. Each duplicated basic block forms a new superblock that is appended to the end of the function. The result is shown in Figure 2.2(b). Note that there are no longer side entrances into the most frequently traversed trace, $< A, B, E, F >$; it has become a superblock.

2.2.2 Superblock Enlarging Optimizations

The first category of superblock ILP optimizations is superblock enlarging optimizations. The purpose of these optimizations is to increase the size of the most frequently executed superblocks so that the superblock scheduler can manipulate a larger number of instructions. It is more likely the scheduler will find independent instructions to schedule at every cycle in a superblock when there are more instructions to choose from. An important feature of superblock enlarging optimizations is that only the most frequently executed parts of a program are enlarged. This selective enlarging strategy keeps the overall code expansion under control [25]. Three superblock enlarging optimizations are described as follows.

Branch Target Expansion. Branch target expansion expands the target superblock of a likely taken control transfer which ends a superblock. The target superblock is copied and appended to the end of the original superblock.

Loop Peeling. Superblock loop peeling modifies a superblock loop (a superblock which ends with a likely control transfer to itself) which tends to iterate only a few times for each loop execution. The loop body is replaced by straight-line code consisting of the first several iterations of the loop.[1] The original loop body is moved to the end of the function to handle executions which require additional iterations. After loop peeling, the most frequently executed preceding and succeeding superblocks can be expanded into the peeled loop body to create a single large superblock.

Loop Unrolling. Superblock loop unrolling replicates the body of a superblock loop which tends to iterate many times. To unroll a superblock loop N times, $N-1$ copies of the superblock are appended to the original superblock. The control transfers in the first $N-1$ loop bodies are adjusted or removed if possible to account for the unrolling.

2.2.3 Superblock Dependence-Removing Optimizations

The second category of superblock ILP optimizations is superblock dependence removing optimizations. These optimizations eliminate data dependences between instructions within frequently executed superblocks, which increases the ILP available to the code scheduler. As a side effect, some of these optimizations increase the number of executed instructions. However, by applying these optimizations only to frequently executed superblocks, the code expansion incurred is regulated. Five superblock dependence removing optimizations are described as follows.

Register Renaming. Register renaming removes anti, output, and control dependences between instruction pairs within a superblock [26].

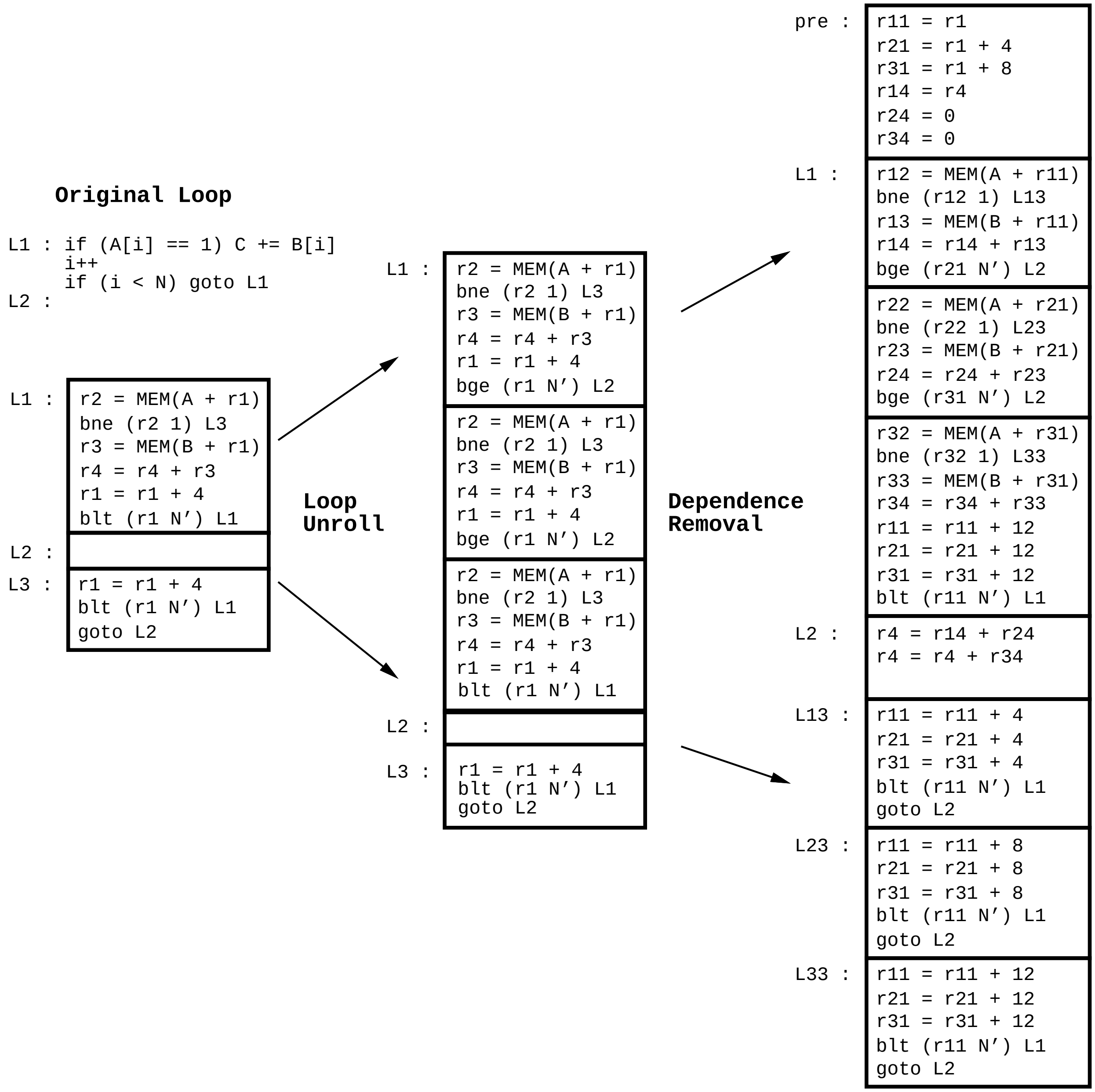
[1] Using the profile information, the loop is peeled its expected number of iterations.

Operation Migration. Operation migration moves an instruction from a superblock where its result is not used to a less frequently executed superblock. By migrating an instruction, all of the data dependences associated with that instruction are eliminated from the original superblock.

Induction Variable Expansion. Induction variable expansion eliminates redefinitions of induction variables within an unrolled superblock loop. Each definition of the induction variable is given a new induction variable, thereby eliminating all anti, output, and flow dependences among the induction variable definitions. However an additional instruction is inserted into the loop preheader to initialize each newly created induction variable. Patch code is inserted if the variable is used outside the superblock.

Accumulator Variable Expansion. Accumulator variable expansion is similar to induction variable expansion. Anti, output, and flow dependences between instructions which accumulate a total are eliminated by replacing each definition of accumulator variable with a new accumulator variable. Unlike induction variable expansion, though, the increment or decrement value is not required to be constant within the superblock loop. Again, initialization instructions for these new accumulator variables must be inserted into the superblock preheader. Also, the new accumulator variables are summed at all superblock exit points to recover the value of the original accumulator variable.

Operation Combining. Operation combining eliminates the flow dependence between a pair of instructions each of which has a constant source operand [27]. The flow dependence is removed by modifying the flow dependent instruction's constant source operand and replacing the flow dependent source operand with an appropriate variable.



```
L1 : if (A[i] == 1) C += B[i]
     i++
     if (i < N) goto L1
L2 :
```

```
L1 : r2 = MEM(A + r1)
     bne (r2 1) L3
     r3 = MEM(B + r1)
     r4 = r4 + r3
     r1 = r1 + 4
     blt (r1 N') L1
L2 :
L3 : r1 = r1 + 4
     blt (r1 N') L1
     goto L2
```

```
L1 : r2 = MEM(A + r1)
     bne (r2 1) L3
     r3 = MEM(B + r1)
     r4 = r4 + r3
     r1 = r1 + 4
     bge (r1 N') L2
     r2 = MEM(A + r1)
     bne (r2 1) L3
     r3 = MEM(B + r1)
     r4 = r4 + r3
     r1 = r1 + 4
     bge (r1 N') L2
     r2 = MEM(A + r1)
     bne (r2 1) L3
     r3 = MEM(B + r1)
     r4 = r4 + r3
     r1 = r1 + 4
     blt (r1 N') L1
L2 :
L3 : r1 = r1 + 4
     blt (r1 N') L1
     goto L2
```

```
pre : r11 = r1
      r21 = r1 + 4
      r31 = r1 + 8
      r14 = r4
      r24 = 0
      r34 = 0
L1 :  r12 = MEM(A + r11)
      bne (r12 1) L13
      r13 = MEM(B + r11)
      r14 = r14 + r13
      bge (r21 N') L2
      r22 = MEM(A + r21)
      bne (r22 1) L23
      r23 = MEM(B + r21)
      r24 = r24 + r23
      bge (r31 N') L2
      r32 = MEM(A + r31)
      bne (r32 1) L33
      r33 = MEM(B + r31)
      r34 = r34 + r33
      r11 = r11 + 12
      r21 = r21 + 12
      r31 = r31 + 12
      blt (r11 N') L1
L2 :  r4 = r14 + r24
      r4 = r4 + r34

L13 : r11 = r11 + 4
      r21 = r21 + 4
      r31 = r31 + 4
      blt (r11 N') L1
      goto L2
L23 : r11 = r11 + 8
      r21 = r21 + 8
      r31 = r31 + 8
      blt (r11 N') L1
      goto L2
L33 : r11 = r11 + 12
      r21 = r21 + 12
      r31 = r31 + 12
      blt (r11 N') L1
      goto L2
```

Figure 2.3 An application of superblock ILP optimizations.

Example. An example to illustrate loop unrolling, register renaming, induction variable expansion, and accumulator variable expansion is shown in Figure 2.3. This example assumes that the condition of the *if* statement is more often true than false. Note that only the frequently executed path is replicated during loop unrolling. Also, there are still dependences across replicated loop bodies due to the increment of loop counter i (r1) and the accumulation into the variable C (r4). Furthermore, the reuse of temporary registers r2 and r3 introduces anti and output dependences across replicated loop bodies. The application of superblock dependence removing optimizations eliminates all these dependences. After all superblock ILP optimizations are applied, the execution of the unrolled loop bodies may be completely overlapped by the superblock scheduler.

CHAPTER 3

COMPILER-CONTROLLED SPECULATION

One of the critical requirements for all compiler transformations is not to introduce program errors. Scheduling errors can occur as a result of illegal re-ordering of operations that change the final result of the program. Obeying data dependences during scheduling will prevent illegal re-ordering. However, global scheduling introduces the possibility of another potential program error - *premature program termination*. This could occur as a result of speculating a divide above a conditional branch. If the conditional branch was either explicitly or implicitly preventing a zero divisor from reaching the divide, the program could terminate. All compiler-controlled speculation models are required to prevent program termination. The models differ in their approaches to detect the exception, report the exception and in some cases actually recover from the exception. The next section provides a detailed explanation of the requirements of compile-time speculation models if they are to accurately report exceptions and recover from exceptions. Section 3.2 classifies existing speculation models by their approach to handling exceptions or preventing these exceptions occurring. Section 3.3 introduces examples of speculation models that fall into each category. Finally, Section 3.4 introduces the IMPACT global scheduler and the heuristics used to permit efficient speculation.

I	Instruction		
1	beq	r5,0,L1	
2	ld	r7,mem(r5)	
3	add	r6,r6,r7	

(a)

I	Instruction		
2	ld	r7,mem(r5)	(S)
1	beq	r5,0,L1	
3	add	r6,r6,r7	

(b)

I	Instruction		
2	ld	r7,mem(r5)	(S)
3	add	r6,r6,r7	(S)
1	beq	r5,0,L1	

(c)

Figure 3.1 Example code segment, (a) un-scheduled code, (b) one speculated instruction, (c) two speculated instructions.

3.1 Accurate Reporting and Recovery from Exceptions

Figure 3.1 illustrates the problems that compiler-controlled speculation models must address to accurately report and recover exceptions. In Figure 3.1(a), instruction I_1 guards against an invalid memory access by I_2. Under normal situations, the address for I_2 is valid. Thus the branch is usually not taken. Instruction I_3 uses the contents of memory to perform a computation. Assuming that the branch has a one cycle latency, the load has a 2 cycle latency and the add has a 1 cycle latency, the shortest latency possible for these three instructions is 4 cycles. Assume that the compiler has renamed the destination registers of I_2 and I_3 so that these instructions cannot corrupt the source operands of instructions on the taken path of the branch instruction. Since the branch is usually not taken, it is feasible to speculate I_2 above I_1 producing the schedule shown in Figure 3.1(b) (an instruction denoted by *(S)* is speculated). This new schedule has a 3 cycles latency and would be a more desirable schedule, if it were legal.

For the schedule in Figure 3.1(b) to be legal, it must not alter the program behavior regardless of the direction the branch takes at run time. If the contents of r5 in I_2 is zero, the load will cause an invalid memory access exception. If the exception is permitted to occur, an error that I_1 was designed to prevent will be falsely reported. One solution to this problem is to delay reporting the exception until the direction of the branch is taken (delayed exception). This requires that the knowledge of the exception must be maintained for later use.

A slightly different problem occurs if the exception caused by I_2 was not an invalid memory access but instead was a page fault. If we delay the exception until after the branch direction is known, we must be able to determine that I_2 caused the exception. Once we know that the branch is not taken, we must re-execute I_2 and permit the exception to occur. This allows the exception to be reported accurately.

The problem becomes even more complicated if the scheduler decided to also move I_3 above I_1 as shown in Figure 3.1(c). If I_3 is permitted to update r6, after an exception by I_2, it will be corrupted. If we re-execute I_3 after the exception from I_2 is resolved in hopes of correcting the error, r6 will still be corrupted. Therefore, we have reached an unrecoverable state.

This example has shown the four problems that must be solved in order for the compiler to accurately report and recover from exceptions resulting speculation of potentially excepting instructions (SPEIs). They include detecting a delayed exception, determining the instruction that caused the exception, protecting source operands until the exception is resolved and recovering from the exception. While the typical exception is viewed as one that caused program termination, this does not need to be the case such as a page fault.

3.2 Classification of Compiler-Controlled Speculation Models

In order to gain greater scheduling freedom, instructions must be allowed to speculate above conditional branches found within a trace or superblock. In some cases, speculation of these instructions can introduce a *run-time errors* that can cause unexpected program termination. Compiler-controlled speculation models can be classified by what the scheduler will do if the

(1) **Avoid Errors** - do not permit the instruction to speculate if it could cause a run-time error [7].

(2) **Ignore Errors** - assumes that the likelihood of the instruction causing an errors is small. Therefore, a non-excepting form of the instruction will be used instead of the excepting form. As a result, if the speculated instruction does cause an error, the error will be lost. This model requires non-excepting forms of each potentially excepting instruction that is speculated [7].

(3) **Resolve Errors** - speculates instructions that could cause a run-time error but uses a combination of compiler and processor extensions extensions to permit corruption of the processor state until it is known whether the instruction should except. Each model also provides a mechanism for recovery in the event that the speculated instruction does except and program execution should resume.

Figure 3.2 Classifications of compiler-controlled speculation models.

an instruction that should be speculated could cause program termination. Figure 3.2, shows three categories that speculation models fall into if this circumstance occurs.

3.2.1 Avoid Errors Models

This scheduling model assumes that incorrect program execution must always be detected. Therefore, the compiler can only speculate instructions that it can guarantee will never cause an exception. In most processors, this prevents speculation of all instructions that reference memory due to potential illegal address references, as well as integer divide and remainder instructions due to potential divide-by-zero errors. Additionally floating point instructions typically are not speculated since they can cause numerous error conditions including underflow and overflow. Using these constraints, this category can be very restrictive to scheduling freedom.

3.2.2 Ignore Errors Models

These models require a non-excepting form of every potentially excepting instruction that can be speculated. Thus, if a potentially excepting instruction should be speculated, it will be replaced by its non-excepting form. The objective of this category is to permit aggressive speculation. The obvious disadvantage is that real program errors could go undetected or cause subsequent errors later in the program. For example, if we speculate a divide above a branch, this model will replace the divide by a non-trapping form of the divide. If the branch branch was implicitly preventing a divide-by-zero, speculating above the branch increases the risk of the divide-by-zero. If we would have always branched around the divide, then the divide-by-zero would never have occurred and hiding the exception using the non-trapping divide causes no harm. However, if there is a programming error, it is possible that the divide-by-zero could happen and we are preventing this error from ever being reported. This could result in some illegal value being propagated to instructions that subsequently were intended to use the result of the divide. This could cause program termination at some later point that could make it difficult to isolate the real problem. The IEEE 754 floating-point standard actually supports this model. The standard requires that floating-point exceptions are sticky. However, while the existence of a trap-handler for these exceptions is recommended, it is not required. The following list provides some examples of architectures that fall into this category in limited circumstances.

(1) Multiflow - non-trapping floating point instructions [28]

(2) Cydra 5 - non-trapping floating point instructions, ability to disable exceptions for memory and arithmetic operations [29].

(3) HP Precision Architecture - non-trapping floating point instructions, non-trapping dereferenced null pointer [30].

(4) Sun SPARC V8 and V9 - non-trapping floating point instructions and non-trapping load instructions [31].

3.2.3 Resolve Errors Models

Just like the prevent errors category, this category assumes that the scheduled program should produce the exact same results as the unscheduled program. However, its ultimate goal is to attain performance results comparable to the ignore errors category. To accomplish this, the speculation model defines compiler and processor extensions that are used to delay reporting an exception until it is determined that the exception should be reported, keep track of the instruction that caused the exception, protect source operands until the exception is resolved and permit recovery from the exception. Thus, potentially excepting instructions may be speculated freely at compile-time as long as the requirements for the recovery mechanism are followed.

3.3 Existing Compiler-Controlled Speculation Techniques

3.3.1 Avoid Error Models

Restricted speculation (formally called restricted code percolation) [7] assumes that correct program execution is always required. Using this model, the compiler can only speculate instructions that will never cause an exception. As such, this model ignores even trivial cases where the context in which the instruction is used, can sometimes indicate if the instruction will cause an exception. Chapter 5 proposes a new speculation model that meets the requirements of this category but uses information that can be gathered intra-procedurally and inter-procedurally to identify potentially excepting instructions that will never cause exceptions.

3.3.2 Ignore Error Models

General speculation (formally called general code percolation) falls under this category. This aggressive model assumes a non-excepting form for every potentially instruction. This model is used as a measure of the maximum performance that speculation models can attain in the absence of register constraints and cache constraints.

3.3.3 Resolve Error Models

This section introduces two speculation models that accurately report exceptions and permit recovery. These speculation models also support aggressive compile-time speculation.

3.3.3.1 Instruction Boosting

Instruction boosting has been proposed for handling exceptions with compiler-controlled speculative execution [32] [33]. The four problems associated with exception detection and recovery are handled with a combination of hardware support (shadow register files) and compiler generated recovery blocks. Detecting delayed exceptions is handled by recording an exception condition raised by a speculative instruction in the appropriate shadow register file. At the excepting instruction's commit point, the contents of the shadow register file are examined to determine if an exception condition exists. The excepting instruction is identified by sequentially re-executing all speculative instructions which are committed by the same branch instruction. The exception condition is therefore regenerated in a sequential processor state. Operands of speculative instructions are preserved by ensuring that speculative instructions do not update the architectural register file until they are committed. Therefore, a speculative instruction may always be re-executed by retrieving its operands from the architectural register file. Finally, recovery is handled with traditional exception recovery techniques since the exception is regenerated in a sequential processor state.

Although boosting provides good support for accurate detection and handling of exceptions for SPEIs, it does so with excessive hardware overhead. The scheme requires multiple copies of register files to implement the shadow registers. The fact that exception recovery requires recovery code blocks also increases code size by about two times, which adds significantly to the pressure on the memory system [33].

3.3.3.2 Sentinel Scheduling

An alternative scheme to enable exception detection and recovery with compiler-controlled speculative execution is sentinel scheduling [34] [35]. Sentinel scheduling is a compiler based technique that requires few changes to the processor architecture. The four problems associated with exception detection and recovery are handled using exception tags added to each architectural register and compiler scheduling and register allocation support. Delayed exceptions are detected by marking exception conditions in the destination register of excepting speculative instructions. The PC of the speculative instruction is also placed in the destination register. Subsequent speculative instructions which use the result of an excepting speculative instruction propagate the PC and exception tag to their destination. A subsequent non-speculative indirect use of the excepting speculative instruction's destination register signals any exception conditions that are present. The excepting instruction is identified by the PC contained in the corresponding register whose exception tag is set.

Source operands for speculative instructions are preserved by ensuring the scheduler and register allocator do not allow any instruction to overwrite a speculative instruction's source

			Current	Speculation	Home
I	Instruction		Block	Distance	Block
1	add	v3,v2,v4	0	0	0
2	add	v1,v2,4	0	0	0
3	beq	v1,0,L1	0	0	0
4	load	v5,mem(v1)	1	0	1
5	store	mem(v5),v2	1	0	1
6	div	v6,v4,v7	1	0	1
7	add	v8,v0,v6	1	0	1
8	bne	v8,v2,L2	1	0	1
9	load	v9,mem(v8)	2	0	2
10	add	v9,v9,4	2	0	2
11	store	mem(v9),v2	2	0	2

Figure 3.3 Original code segment.

operands before a non-speculative instruction checks the exception condition of the speculative instruction. Finally, recovery is performed by setting the PC to the excepting instruction's PC and re-executing all speculative instructions until the check instruction is reached.

An example code segment to illustrate speculative execution with sentinel scheduling is shown in Figure 3.3. The example consists of three basic blocks in which the compiler expects control flow to be sequential through the blocks. Furthermore, load and div instructions are assumed to be potentially excepting. The code segment after scheduling is shown in Figure 3.4. Speculative instructions are denoted by (S). With sentinel scheduling, the scheduler ensures that there is a non-speculative instruction to check the exception tag of each PEI in the PEI's home block. For example, I_8 checks if an exception occurred for I_6.[1] In order to protect the source operands, the lifetimes of all source operands for speculative instructions are extended to the non-speculative checking instruction. For example, the lifetimes of v4 and v7 (source operands of I_6) are extended to I_8. Also, the scheduler must not schedule any instructions

[1] An exception condition for I_6 will mark an exception in v6. I_7 will propagate the exception to its destination, v8, since it is also a speculative instruction. Finally, the use of v8 by I_8 will detect the exception condition.

				Current	Speculation	Home
I	Instruction			Block	Distance	Block
6	div	v6,v4,v7	(S)	0	1	1
1	add	v3,v2,v4		0	0	0
7	add	v8,v0,v6	(S)	0	1	1
2	add	v1,v2,4		0	0	0
9	load	v9,mem(v8)	(S)	0	2	2
4	load	v5,mem(v1)	(S)	0	1	1
10	add	v10,v9,4	(S)	0	2	2
3	beq	v1,0,L1		0	0	0
5	store	mem(v5),v2		1	0	1
8	bne	v8,v2,L2		1	0	1
11	store	mem(v10),v2		2	0	2
12	mov	v9,v10		2	0	2

Figure 3.4 Code segment after sentinel scheduling, before register allocation.

which overwrite a speculative instruction's source operands. Therefore, the destination of I_{10} must be renamed to v10 to prevent v9 from being overwritten. The code segment after register allocation is shown in Figure 3.5. In this example, a total of 10 physical registers are required to ensure exception detection and recovery are possible for all speculative instructions.

3.4 Scheduling in the IMPACT Compiler

The instruction scheduler used in the IMPACT compiler performs both local and global instruction scheduling. Local scheduling is performed on basic blocks and permits some limited reordering of instructions as well as filling of branch delay slots [36]. While the benefits of this level of scheduling may be adequate for single-issue microprocessors, it has been shown to be inadequate when using non-numeric benchmarks for wide-issue processors [1, 2, 7]. This occurs because there are typically few instructions in basic blocks and only a small number of independent computation chains. In order to gain greater optimization and scheduling freedom,

I	Instruction			Current Block	Speculation Distance	Home Block
6	div	r9,r2,r6	(S)	0	1	1
1	add	r5,r4,r2		0	0	0
7	add	r7,r0,r9	(S)	0	1	1
2	add	r3,r4,4		0	0	0
9	load	r8,mem(r7)	(S)	0	2	2
4	load	r1,mem(r3)	(S)	0	1	1
10	add	r9,r8,4	(S)	0	2	2
3	beq	r3,0,L1		0	0	0
5	store	mem(r1),r4		1	0	1
8	bne	r7,r4,L2		1	0	1
11	store	mem(r9),r4		2	0	2
12	mov	r8,r9		2	0	2

Figure 3.5 Code segment after sentinel scheduling and register allocation.

superblocks were developed to permit optimization and scheduling of multiple basic blocks with similar execution paths together [4]. Section 3.4.2 will describe the scheduling heuristics used in the impact scheduler. Section 3.4.3 will provide experimental results showing the benefits of the IMPACT instruction scheduler.

3.4.1 Overview of the IMPACT Scheduler

Acyclic scheduling is accomplished by performing two passes over each basic block or superblock. The first pass, pre-pass scheduling, is performed prior to register allocation. As such, the scheduler has the greatest scheduling freedom since no artificial data dependences resulting from processor register constraints have been created. The second pass, post-pass scheduling, is performed after register allocation. This pass is used to resolve any scheduling changes that may occur resulting from the insertion of spill code, as well as caller and callee save code required to allocate the virtual registers to the physical registers. This pass is also used to fill branch delay slots [36].

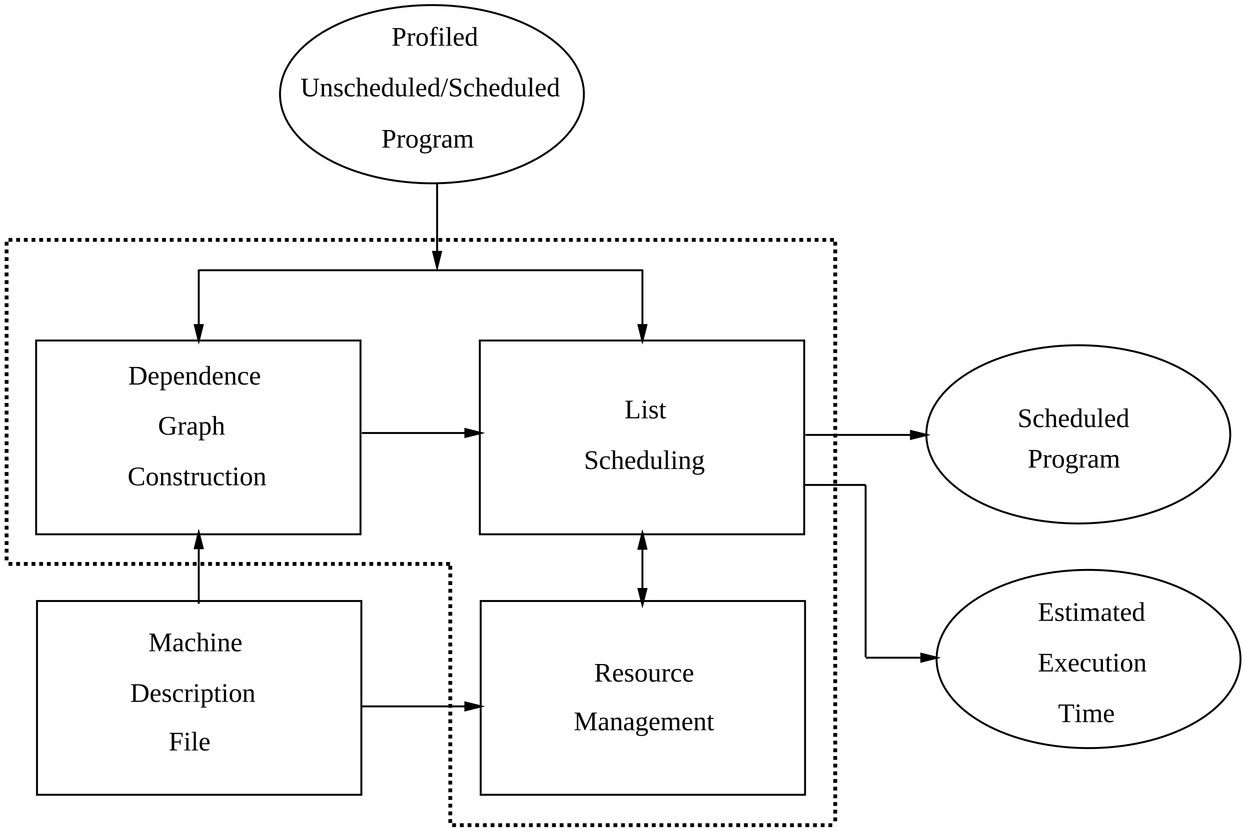


Figure 3.6 Organization of the IMPACT Instruction Scheduler.

Figure 3.6 shows the organization of the IMPACT scheduler. The blocks inside the dotted line belong to the scheduler. Each pass of the scheduler is provided the entire program. The dependence graph is constructed from the natural data dependences visible within a function. Memory dependences are added between instructions with true memory dependences as well as from instructions with ambiguous memory addresses to all other memory instructions. The IMPACT scheduling model does not permit re-ordering of branches or subroutine calls. Thus, control dependences are added to prevent any such re-ordering. Control dependences are added from instructions above a branch to the branch if their destination registers are in the live-out of the branch. This prevents migration of this instructions to other scheduled basic blocks or superblocks as well as incorrect program execution. Similarly, control dependences are added from instructions below the branch to the branch if their destination registers are in the live-out of the branch. This prevents corruption of the live registers that could lead to incorrect program execution. Finally, the speculation model defines the conditions under which potentially excepting instruction may be speculated above conditional branches. Based upon the requirements of the speculation model, additional control dependences are added to prevent illegal speculation.

Once the dependence graph is constructed, list scheduling is performed on a basic block or superblock basis. The first step of the list scheduler is to prioritize the instructions. This heuristic is discussed in Section 3.4.2. After the instructions are prioritized, they are placed into one of two queues. Instructions that have no incoming dependences located within the block are placed in the *ready queue* in sorted order from highest priority to lowest priority. All other instructions are placed in the *not ready queue*.

Instructions are then removed from the ready queue starting with the highest priority instruction. The scheduler then queries the resource manager to determine if the instruction can be scheduled in the current cycle. The resource manager maintains the scheduled state of the block for each cycle. Legal assignments are dependent upon the resource constraints defined by the machine description file. The resource manager determines the location in the cycle that an instruction may be scheduled and allocates the necessary resources. Thus, the priority of an instruction can be over-ridden by the scheduling constraints. This prevents scheduling an instruction that could lead to processor interlock [36]. An instruction that can not be scheduled in a given cycle is moved to the *can't schedule queue*, which exposes the next instruction in the ready queue.

Once an instruction is scheduled, the dependence constraints are removed from any dependent instructions. Removing of these dependence constraints requires specifying the earliest cycle that an instruction may be scheduled in order to maintain accuracy. Only the latest legal scheduling time is maintained for an instruction to ensure that all dependences are maintained. After each instruction is scheduled, instructions with no remaining dependences are moved into either the *pending ready queue* where they will remain until the desired scheduling cycle, or into the ready queue if the scheduling cycle has been reached.

As long as the ready queue contains instructions, the schedule time will advance by only one cycle after all possible instructions are scheduled for a given cycle. At the start of each cycle, instructions can be moved from the pending ready queue to the ready queue. If there are no instructions in the ready queue, time will advance to the earliest cycle that an instruction can be scheduled in the pending ready queue [2]. After all instructions are scheduled, the issue time,

[2] In order to legally schedule for a VLIW processor, the cycles between the last scheduled instruction and the next ready instruction will be padded with no-op instructions.

issue slot and completion time of each instruction is marked and the instructions are inserted, following the scheduled order, back into the basic block or superblock.

The scheduler can also produce an estimated execution time for a function based upon the profile information provide, the static branch probabilities and the instruction schedule. Two execution estimates will be generated. The first, gives the best case execution time which assumes that instructions whose completion time overlaps the beginning of subsequent blocks will not cause any processor interlock conditions . The second estimate assumes that the worst case scenario will occur and all instructions whose completion time overlaps the beginning of subsequent blocks will cause interlock. Experimental analysis has shown that these results are within 3 to 5 percent of simulation results when assuming perfect caches. These approximate estimates provide a fast method to evaluate the benefits of various compiler transformations. For this research, all final numbers are generated using simulation.

3.4.2 Superblock Scheduling

Superblock scheduling is complicated by speculation decisions that must be made. Speculating an instruction above a frequently taken branch may delay the branch and result in poor performance. However, speculating an instruction above an infrequently taken branch can permit the overlap of its execution with earlier instructions which can result in shorter program execution times. Since each branch may have different probabilities of being taken, the advantages or disadvantages associated with speculating instructions above branches are not equal.

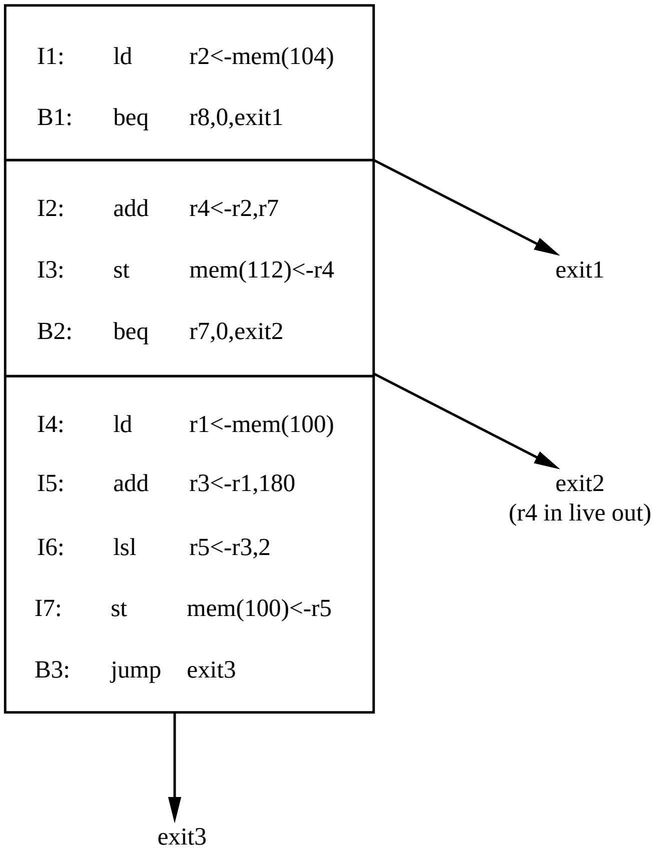
Fisher proposed a heuristic called **speculative yield** that can be used to help assign instruction priorities in the presence of conditional branches [37]. This heuristic uses branch probabilities to determine the benefit of speculating an instruction from either path of a conditional branch. Fisher's use of this heuristic assumes the scheduling is being performed as the trace is being generated. In contrast, superblock formation and optimization is performed earlier in the compilation process. This reduces the complexity of the scheduling decisions and permits simplification of the heuristic to a static heuristic as shown in equation 3.1.

$$Prio(I_j) = \sum_{k=1}^{n} (Prob(path_k) * length(path_k)) \tag{3.1}$$

By using profile information [25] or static branch prediction heuristics [38, 39] the probabilities for the paths can be estimated[3]. The lengths of the paths are computed using the dependence distances from an instruction to its last use. As this equation shows, each path that an instruction reaches contributes to its priority.

Figure 3.7 shows an example superblock that will be a working example to demonstrate the use of the speculative yield heuristic. For the remaining discussions on this example, assume the latency on a load is one cycle and that all other instructions have single cycle latencies. Using this information, the dependence graph in Figure 3.8 can be generated. This figure shows the flow, control and anti dependences found in the superblock example. The horizontal dotted lines indicate not only a dependence but a required ordering that must be maintained to ensure correctness in the case of the store instruct I3 and to prevent instructions from be scheduled outside the superblock. From the dependence graph, we can see that there are actually three paths that may occur in parallel. Given a processor that can issue any combination of three or more instructions per cycle, this example can be scheduled in 5 cycles. To make this example

[3] Fisher and Freudenberger showed that previous program runs provide highly accurate estimates for predicting conditional branch direction [40].



```
I1:   ld     r2<-mem(104)
B1:   beq    r8,0,exit1

I2:   add    r4<-r2,r7
I3:   st     mem(112)<-r4
B2:   beq    r7,0,exit2

I4:   ld     r1<-mem(100)
I5:   add    r3<-r1,180
I6:   lsl    r5<-r3,2
I7:   st     mem(100)<-r5
B3:   jump   exit3
```

Figure 3.7 An example superblock.

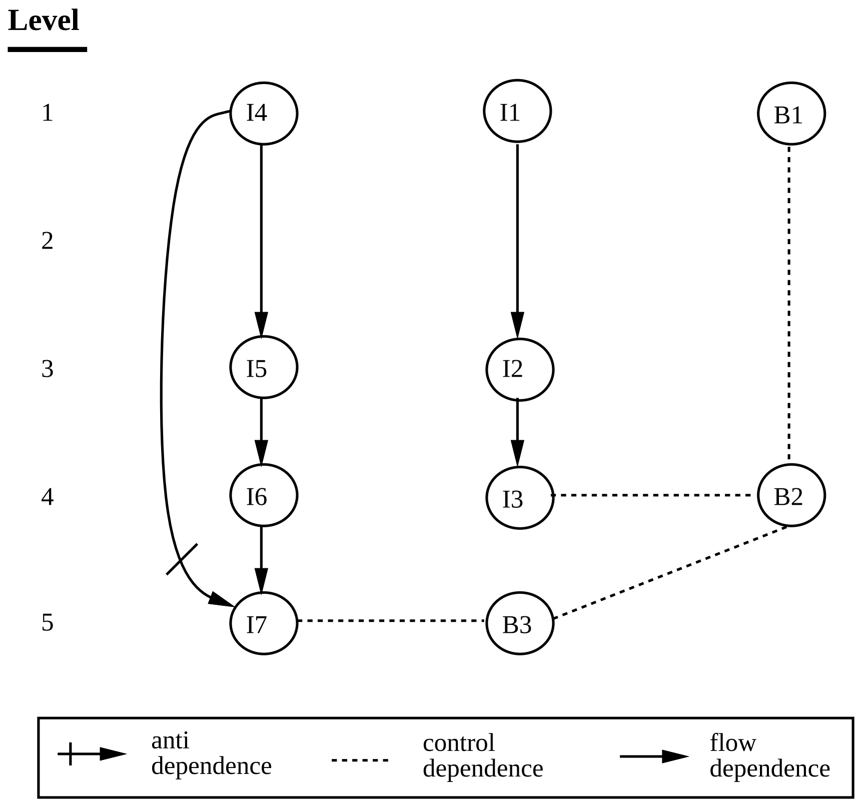


Figure 3.8 Corresponding dependence graph for example superblock.

Algorithm Compute_Static_Priority: Given a superblock or basic block, compute static priorities for instructions based upon the paths that the instructions reach.

```
levelize dependence graph for block

determine path length of exit arcs

compute maximum path lengths from each instruction to each of its exit arcs

compute static priorities using path lengths and branch probabilities
```

Figure 3.9 The Compute_Static_Priority algorithm determines the instruction priorities for each instruction.

more interesting, we will assume for the remaining discussion that the target processor can issue any combination of **two** instructions per cycle.

Figure 3.9 shows the algorithm that will be used to compute the static scheduling priorities using the speculative yield heuristic. As the figure shows, four steps are performed when computing the static scheduling priorities for instructions. The first step is to levelize the dependence graph as shown in Figure 3.8. This will show the earliest issue time for all instruction given unlimited resources but obeying instruction latencies and dependences.

Step two determines the lengths that will be used for the exit arcs which are represented by the taken path of each branch. This step is performed to ensure that the probability of a branch being taken is of equal importance to the longest path through the block. Without this step, a highly taken branch could be easily delayed by a long computation path through block. For example if the first branch in a superblock is taken 99 percent of the time and there are no instructions that need to be scheduled before the branch, it could be scheduled in the first cycle ensuring the minimum path through the superblock 99 percent of the time. Now assume an extreme case where an execution path through the same superblock is 200 cycles long. Since these instructions only need to be executed .01 percent of the time, they should all be scheduled

	Max Path Length		
Instruction	**B1**	**B2**	**B3**
I1	-	5	4
B1	5	2	1
I2	-	3	2
I3	-	2	1
B3	-	2	1
I4	-	-	5
I5	-	-	3
I6	-	-	2
I7	-	-	1
B3	-	-	1

Figure 3.10 Maximum path lengths from starting instruction to branch.

after the branch. However, the instructions for the first 100 cycles will have priorities greater than .99 based upon their dependence heights. As such, they would be scheduled before the branch and increase the execution time of the most frequent path. The path length is computed by taking the maximum level in the graph, in this case 5, and subtracting the level of the branch and then adding 1. The addition of one ensures that zero priorities only occur for instructions that follow branches that are always taken. By applying this step to the graph in Figure 3.8, the initial path length of B1 is 5, the initial path length of B2 is 2 and the initial path length of B3 is 1.

Step three is used to compute the maximum path lengths from each instruction to their corresponding exit arcs. The maximum path lengths are computed by starting with the path length at each exit arc and traversing the graph from leaf to root. The maximum path length that reaches a node is maintained. Figure 3.10 shows the maximum path lengths that are computed from the dependence graph using the initial path lengths shown in the left-most column for each branch.

The fourth step is to compute the static priorities using the speculative yield formula shown in Equation 3.1. In order to show the full effects of this, we are going to assume three groups

	Priority
Instr	B1 B2 B3
I1	(0 * 0.2) + (5 * 0.2) + (4 * 0.6) = 3.4
B1	(5 * 0.2) + (2 * 0.2) + (1 * 0.6) = 2.0
I2	(0 * 0.2) + (3 * 0.2) + (2 * 0.6) = 1.8
I3	(0 * 0.2) + (2 * 0.2) + (1 * 0.6) = 1.0
B2	(0 * 0.2) + (2 * 0.2) + (1 * 0.6) = 1.0
I4	(0 * 0.2) + (0 * 0.2) + (5 * 0.6) = 3.0
I5	(0 * 0.2) + (0 * 0.2) + (3 * 0.6) = 1.8
I6	(0 * 0.2) + (0 * 0.2) + (2 * 0.6) = 1.2
I7	(0 * 0.2) + (0 * 0.2) + (1 * 0.6) = 0.6
B3	(0 * 0.2) + (0 * 0.2) + (1 * 0.6) = 0.6

	Schedule	
0	I1	I4
1	B1	
2	I2	I5
3	I6	I3
4	B2	I7
5	B3	

Figure 3.11 Instruction priorities and schedule assuming branch probabilities are 20% for B1, 20% for B2 and 60% for B3.

	Priority
Instr	B1 B2 B3
I1	(0 * 0.2) + (5 * 0.6) + (4 * 0.2) = 3.8
B1	(5 * 0.2) + (2 * 0.6) + (1 * 0.2) = 2.4
I2	(0 * 0.2) + (3 * 0.6) + (2 * 0.2) = 2.2
I3	(0 * 0.2) + (2 * 0.6) + (1 * 0.2) = 1.4
B2	(0 * 0.2) + (2 * 0.6) + (1 * 0.2) = 1.4
I4	(0 * 0.2) + (0 * 0.6) + (5 * 0.2) = 1.0
I5	(0 * 0.2) + (0 * 0.6) + (3 * 0.2) = 0.6
I6	(0 * 0.2) + (0 * 0.6) + (2 * 0.2) = 0.4
I7	(0 * 0.2) + (0 * 0.6) + (1 * 0.2) = 0.2
B3	(0 * 0.2) + (0 * 0.6) + (1 * 0.2) = 0.2

	Schedule	
0	I1	B1
1	I4	
2	I2	
3	I3	B2
4	I5	
5	I6	
6	I7	B3

Figure 3.12 Instruction priorities and schedule assuming branch probabilities are 20% for B1, 60% for B2 and 20% for B3.

	Priority
Instr	B1 B2 B3
I1	(0 * 0.6) + (5 * 0.2) + (4 * 0.2) = 1.8
B1	(5 * 0.6) + (2 * 0.2) + (1 * 0.2) = 3.6
I2	(0 * 0.6) + (3 * 0.2) + (2 * 0.2) = 1.0
I3	(0 * 0.6) + (2 * 0.2) + (1 * 0.2) = 0.6
B2	(0 * 0.6) + (2 * 0.2) + (1 * 0.2) = 0.6
I4	(0 * 0.6) + (0 * 0.2) + (5 * 0.2) = 1.0
I5	(0 * 0.6) + (0 * 0.2) + (3 * 0.2) = 0.6
I6	(0 * 0.6) + (0 * 0.2) + (2 * 0.2) = 0.4
I7	(0 * 0.6) + (0 * 0.2) + (1 * 0.2) = 0.2
B3	(0 * 0.6) + (0 * 0.2) + (1 * 0.2) = 0.2

	Schedule	
0	B1	I1
1	I4	
2	I2	
3	I5	I3
4	B2	I6
5	I7	B3

Figure 3.13 Instruction priorities and schedule assuming branch probabilities are 60% for B1, 20% for B2 and 20% for B3.

of priorities for the branches from the example. In each example, one branch is assumed to be taken 60 percent of the time and the other two branches are taken 20 percent of the time each. Figures 3.11 through 3.13 show the resultant priorities for each instruction and their schedules using a two issue processor. The important point of each figure is that in all cases, the most frequently taken branch was issued in the earliest possible cycle given the resource constraints of a two issue processor.

3.4.3 Experimental Evaluation and Analysis

This section will present experimental results that show the benefits of the IMPACT scheduler using the scheduling heuristic discussed in the previous section. The benchmarks used in this study are the 15 non-numeric programs shown in Table 3.1. The benchmarks consist of the 6 non-numeric programs from the SPECint92 suite and 9 other commonly used non-numeric programs. Each of the benchmarks were aggressively optimized and scheduled assuming the

Benchmark	Benchmark Description
008.espresso	truth table minimization
022.li	lisp interpreter
023.eqntott	boolean equation minimization
026.compress	compress files
072.sc	spreadsheet
085.cc1	GNU C compiler
cccp	GNU C preprocessor
cmp	compare files
eqn	format math formulas for troff
grep	string search
lex	lexical analyzer generator
qsort	quick sort
tbl	format tables for troff
wc	word count
yacc	parser generator

Table 3.1 Benchmarks.

Function	Latency	Function	Latency
Int ALU	1	FP ALU	2
memory load	2	FP multiply	2
memory store	1	FP divide(single-precision)	8
branch	1 / 1 slot	FP divide(double-precision)	15

Table 3.2 Instruction latencies.

general speculation model. The speculative yield heuristic was used to assign static scheduling priorities. The benchmarks were then scheduled for issue rates 1, 2, 4, 8 and infinite issue assuming an in-order issue superscalar processor with register interlocking. The processor is assumed to have uniform functional units, 1 branch delay slot, and the instruction set of the HP PA-RISC processor. The instruction latencies assumed are those of the HP PA-RISC 7100 (see Table 3.2). In order to show the full benefits or limitations of the scheduling heuristic, the code was register allocated with the IMPACT register allocator using infinite registers [19]. In addition, the execution times were generated using perfect instruction and data caches. The data cache effects will be discussed in Chapter 6.

Figure 3.14 presents the speedups of issue 1, 2, 4, 8 and infinite issue scheduled using the speculative yield heuristic with the general speculation model over the respective benchmarks

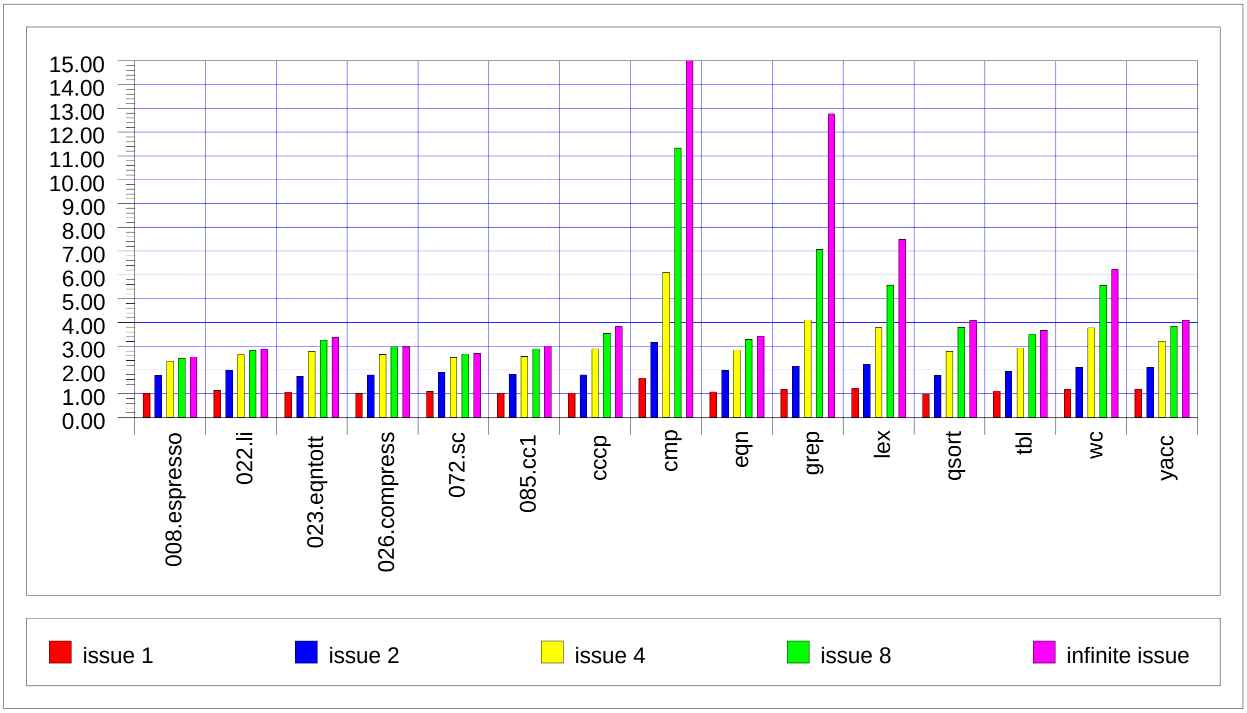


Figure 3.14 Speedup results from the speculative yield scheduling heuristic.

at issue 1 with no speculation using code generated for the restricted speculation model. Since infinite issue will permit scheduling of all possible instructions each cycle, it is unaffected by the scheduling heuristic. As such, these numbers will show the relative merits of the scheduling heuristic. From the graph, all of the single issue results and a number of the higher issue rates show super-linear speedup. While the no speculation and the general speculation code bases were both aggressively optimized using traditional optimizations [41] as well as superblock optimizations, general speculation has a clear advantage over no speculation since it is an ignore errors model. In particular, the model is able to remove invariant operations from loops that might potentially introduce run-time errors. Since the restricted model must make more conservative assumptions during optimization, the number of instructions executed for general speculation is actually smaller.

Additionally, the figure shows, that the 8-issue results are comparable to the infinite issue results for most benchmarks. In these cases, the natural dependences found in the superblock code was the limiting factor on potential performance not the speculative yield heuristic. However, cmp, grep, and lex show a noticeable increase over the 8-issue results [4]. In each of these cases, the superblock optimizations created significantly more instruction level parallelism than could be supported by an 8-issue superscalar processor. The degree of difference correlated directly to the importance of highly parallel segments of the benchmarks. For example, the performance of cmp is dominated by one superblock loop that accounts for 92 percent of the execution time which accounts for the difference in the 8-issue and infinite issue results. After applying the superblock optimizations, the loop contained 66 instructions. The infinite issue processor was able to execute each iteration of this loop in four cycles. One of the cycles even contained 37 independent branch instructions. Given the resource constraints of an 8-issue processor the minimum number of cycles for this same loop is 9 cycles. The speculative yield heuristic resulted in priorities that permitted the loop to be scheduled in the minimum number of cycles.

Figure 3.15 shows the actual instructions per cycle that were attained as a result of the global scheduling using the speculative yield priority heuristic using the general speculation model. Two important points can be seen from this figure. First, the speculative yield heuristic is not also beneficial for high issue rate processors, but also for low issue rates. In particular, the results show that single and dual issue processor results are almost saturated for all of the benchmarks. Secondly, by comparing the bars for issue 8 with infinite issue, it can be noted

[4] For readability, the speedup results for cmp were clipped at 15. The actual speedup attained over single issue no speculation was 21.54.

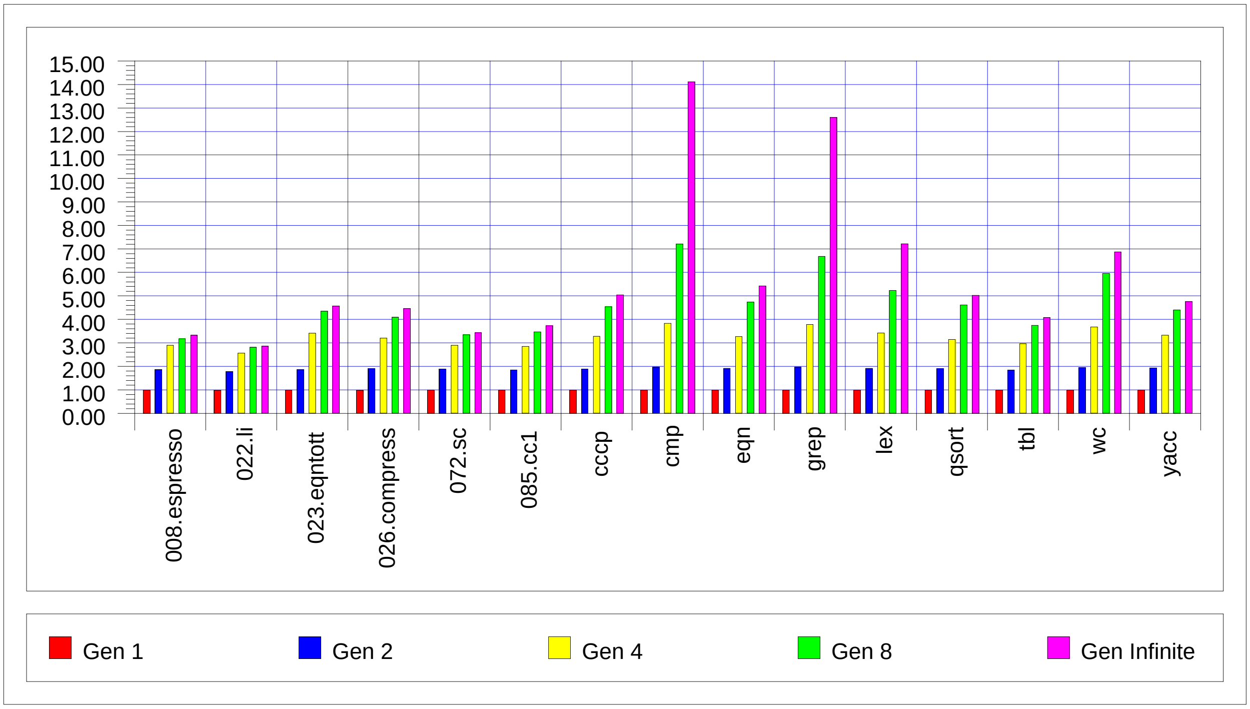


Figure 3.15 Instructions per cycle results from the speculative yield heuristic.

that the IPC's for almost all of the benchmarks are very close. The only noticeable exceptions are for cmp, grep and lex which were discussed previously.

CHAPTER 4

WRITE-BACK SUPPRESSION

This chapter introduces *write-back suppression* (WBS), an architectural support for compiler-controlled speculative execution. WBS makes use of static program information and a set of architectural features to accurately detect and report exceptions for compiler-scheduled speculative instructions. This technique is based on two main concepts: delay the exception for a SPEI until its execution is confirmed, and prevent corruption of the source operands of instructions by systematically suppressing updates to the register file after retiring an excepting SPEI. This implementation of WBS assumes superblocks are used as the means of exposing ILP [4].

4.1 Overview of Write-back Suppression

A few key terms can be defined using the examples in Figures 3.3 and 4.1. The *home block* of an instruction is the basic block where the instruction was located prior to scheduling. Since a superblock does not permit side-entrances, every home block can be assigned a unique identifier by numbering the first home block 0 and incrementing the home block number by one for each fall through branch. In Figure 3.3, instructions I_1 through I_3 belong to home block 0, I_4 through I_8 belong to home block 1 and I_9 through I_{11} belong to home block 2. The *current block* of an instruction is the basic block which the instruction is located in after scheduling. In Figure 4.1, I_6 has been speculated into current block 0. The *speculation distance*

I	Instruction			Current Block	Speculation Distance	Home Block
6	div	v6,v4,v7	(S)	0	1	1
1	add	v3,v2,v4		0	0	0
7	add	v8,v0,v6	(S)	0	1	1
2	add	v1,v2,4		0	0	0
9	load	v9,mem(v8)	(S)	0	2	2
4	load	v5,mem(v1)	(S)	0	1	1
10	add	v9,v9,4	(S)	0	2	2
3	beq	v1,0,L1		0	0	0
12	check			1	0	1
5	store	mem(v5),v2		1	0	1
8	bne	v8,v2,L2		1	0	1
13	check			2	0	2
11	store	mem(v9),v2		2	0	2

Figure 4.1 Code segment after write-back suppression scheduling, before register allocation.

of an instruction is the number of branches that an instruction was speculated beyond during scheduling.[1] In Figure 4.1, I_6 has a speculation distance of 1.

The instruction schedule shown in Figure 4.1 contains only minor modifications from the schedule in Figure 3.4 that was generated by sentinel scheduling. This example introduces a new instruction called *check* which confirms and reports the delayed exceptions from SPEIs. As the figure shows, only one check instruction needs to be placed in a home block even though multiple PEIs were speculated from that block (I_{12} will report the exception for both I_6 and I_4). No check instructions are required for a home block that has had no PEIs speculated from it. If an exception occurs to either I_6 or I_4, it will be recorded at a location unique to I_{12} along with the PC of the appropriate excepting instruction. Subsequently I_{12} will verify the existence of an exception, report the exception for the excepting instruction, and initiate recovery.

[1] speculation distance = home block - current block.

As discussed in Section 3.1, in order for the delayed excepting instruction to accurately recover to the correct processor state, its source operands must be protected until the exception is confirmed. In WBS, the home block of the excepting instruction is used to suppress register file updates for any subsequent instructions from the same or later home blocks. In Figure 4.1, if I_6 excepts, it will prevent I_7, I_9, I_4 and I_{10} from updating the register file. Instructions I_1, I_2 and I_3 will be permitted to execute normally to produce the correct processor state if the branch at I_3 is taken. If the branch I_3 is not taken, its check instruction (I_{12}) will make sure that the exception from I_6 is reported and then re-execute I_6, I_7, I_9, I_4 and I_{10} as part of the recovery phase. In addition to suppressing the updates to the register file of speculated instructions, exceptions will also be suppressed. Thus, if I_9 or I_4 were to also except after I_6, their exceptions would be suppressed until the recovery phase of I_6.

If I_9 and I_4 both except when no other exceptions are pending, we have *nested exceptions.* This occurs because I_9 (from home block 2) can only suppress I_{10} (from home block 2) and will not suppress I_4 (from home block 1). Instruction I_4 also suppresses I_{10} and will ultimately enter its recovery phase via I_{12}. However, one can no longer simply re-execute I_4 and I_{10} to recover from an exception for I_4. Recall that there was a nested exception condition; re-executing I_{10} during I_4's recovery phase will produce an undesirable program state since the exception for I_9 also wants to suppress the register file update for I_{10}. Correct suppression of updates can be accomplished by re-executing only those instructions that belong to I_4's suppression set of instructions but do not belong to I_9's suppression set of instructions. In this example, no instructions will be re-executed during I_4's recovery phase. When the check instruction I_{13} is encountered, it will report the exception for I_9. The recovery will be done by re-executing I_9 and all the instructions in its suppression set, namely, I_{10}.

				Current	**Speculation**	**Home**
I	**Instruction**			**Block**	**Distance**	**Block**
6	div	r1,r2,r1	(S)	0	1	1
1	add	r5,r6,r2		0	0	0
7	add	r1,r0,r1	(S)	0	1	1
2	add	r4,r6,4		0	0	0
9	load	r3,mem(r1)	(S)	0	2	2
4	load	r2,mem(r4)	(S)	0	1	1
10	add	r3,r3,4	(S)	0	2	2
3	beq	r4,0,L1		0	0	0
12	check			1	0	1
5	store	mem(r2),r6		1	0	1
8	bne	r1,r6,L2		1	0	1
13	check			2	0	2
11	store	mem(r3),r6		2	0	2

Figure 4.2 Code segment after write-back suppression scheduling and register allocation.

This example demonstrates the hierarchy of delayed exception handling in WBS. The instruction from the earliest home block will always be resolved first. Multiple exceptions from the same home block will be handled based upon their static order in the scheduled program. All exceptions are handled from earliest home block to latest home block.

Suppressing the updates to the register file has several benefits. First, it eliminates the need for special hardware to maintain the history of the state for the source operands. It also provides the ability to re-use registers amongst speculated instructions. As Figure 4.2 shows, I_6 is able to use one of its own source operands as a destination and still recover correctly. WBS ensures that if an exception occurs, the source operand will not be corrupted. This is not possible with sentinel scheduling since it must maintain a history of the state of all source operands until the sentinel instruction is reached. WBS is able to correctly execute the program segment in Figure 4.2 using only 7 registers while sentinel scheduling requires 10 registers. As this example shows, WBS can reduce the register pressure which will result in less spill code.

4.2 Architectural Extensions

WBS requires several extensions to the architecture. These extensions are broken down into two groups: instruction extensions, and extensions to support suppression of register file updates and recovery. Each of these extensions will be discussed in light of how they support the requirements of WBS as described in Section 4.1.

4.2.1 Instruction Requirements

Each instruction opcode will be augmented with a *k-bit* speculation distance field specifying the number of branches that an instruction has been speculated above. A value of zero in this field indicates that an instruction is not speculated. This field is used by the scheduler to convey the static home block numbers of each instruction to the suppression circuit. A k-bit speculation distance will permit an instruction to be speculated above K=2^k-1 branches.

The *check* instruction is added to the instruction set as a means of reporting and initiating recovery for an exception from a SPEI. This instruction uses its home block number to determine if a SPEI has generated an exception.

4.2.2 Hardware Requirements

Figure 4.3 depicts the hardware extensions required to suppress register file updates for a processor architecture that supports in-order retire. First, a k-bit register specifying the current block along with logic to support increment by one is added. If a branch instruction is not taken, it causes the current_block register to be incremented by one. Otherwise, it will reset the suppression hardware to prevent an exception from propagating into another superblock.

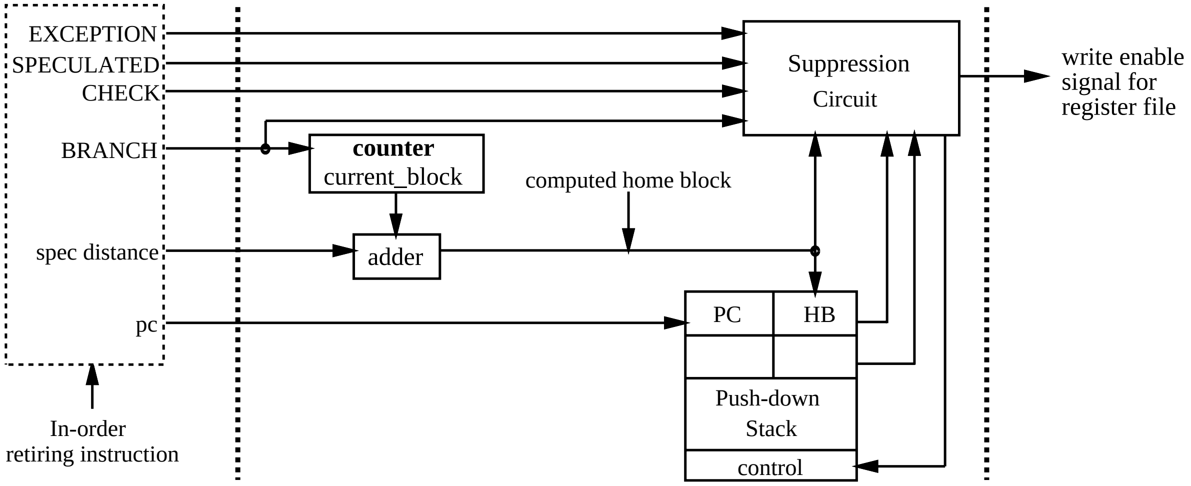


Figure 4.3 Write-back suppression hardware.

Second, a k-bit adder is provided to compute the instruction's home block based upon the current_block and the speculation distance of the retiring instruction.

Next, a k-entry push-down stack is implemented with shift registers to maintain the home block and PC of excepting instructions. Finally, a suppression circuit is added that is controlled by the CHECK and SPECULATED flags of the retiring instruction. The CHECK flag indicates that the current instruction is a check instruction. The SPECULATED flag indicates that the current instruction has been speculated away from its home block. These flags along with the excepting home block numbers from the push-down stack are used by the suppression circuit to determine when it is legal to write to the register file.

The suppression circuit shown is composed of a k-bit comparator with some additional combinatorial logic that implements the state machine shown in Figure 4.4. This circuit is responsible for determining if the instruction currently being retired is allowed to update the register file. The CHECK, SPECULATED and EXCEPTION flags along with the home block numbers from the push-down stack control transition in the state machine between the *normal*

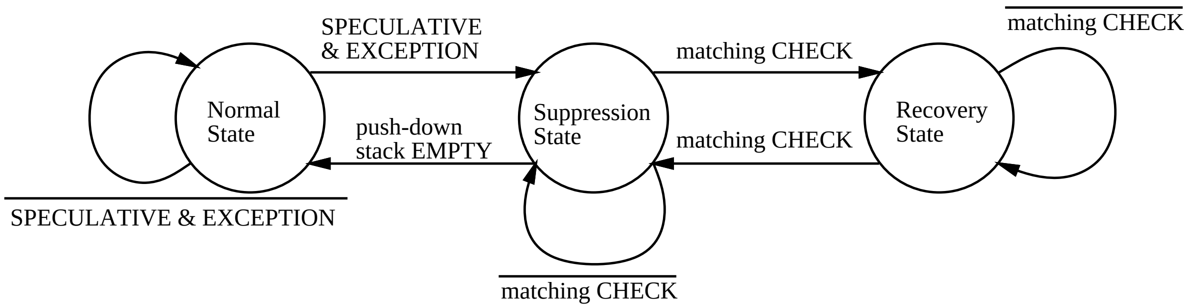


Figure 4.4 Write-back suppression state machine.

state, *suppression state*, and the *recovery state*. All instructions are permitted to update the register file during the normal state.

Starting in the normal state, an instruction whose SPECULATED and EXCEPTION flags are set will place the WBS state machine into the suppression state, and cause the computed home block and its PC to be pushed onto the top of the *push-down stack*. Its update to the register file will be suppressed.

During the suppression state, the computed home block of the retiring instruction and the excepting home block number on the top of the suppression stack will be provided to the suppression circuit to determine if an instruction should update the register file. The suppression circuit will set the *write enable* line for the register file if the computed home block is less than the home block number on the top of the stack, otherwise the write will be disabled. This protects the source operands as required by WBS.

As mentioned in Section 4.1, a nested exception occurs when an excepting SPEI retires with a computed home block above the current excepting home block. This is maintained by pushing the new higher priority exception onto the top of the suppression stack.

If the retiring instruction is a check and its computed home block is the same as the home block number on the top of the push-down stack, a matching CHECK condition is generated.

The state machine will then transition into the recovery state. The PC from the push-down stack will be provided to the fetch stage of the processor to begin recovery. The next time that the instruction associated with the excepting PC is encountered, it will be permitted to except.

During the recovery state, all instructions whose computed home block is less than the excepting home block number on the top of the push-down stack will be prohibited from updating the register file ensuring that these instructions will not be re-executed. Any instructions whose computed home block is greater than or equal to the home block number in the second entry of the push-down stack will also be inhibited from updating the register file to correctly address the nested exception problem. These instructions will be executed once the check for the second entry in the push-down stack is encountered. All other instructions will be permitted to update the register file. The use of excepting home block numbers found in the top two entries of the push-down stack ensures that instructions will only be permitted to update the register file one time. Once the check instruction which initiated the recovery state is encountered again, the top-most entry of the push-down stack will be popped and the WBS state machine will transition back to the suppression state. If the suppression stack is empty, the state machine will transition back to the normal state.

4.2.3 Hardware Cost Estimates

The push-down stack represents the greatest portion of the hardware costs associated with WBS. Assuming that each excepting PC requires 32 bits and the maximum speculation distance, K, is 7 branches (refer to Section 4.5.2.1) requiring 3-bits for storage, the push-down stack hardware would require seven 35-bit shift registers to maintain the exception history. In addition to the push-down stack, WBS requires a 3-bit register to maintain the current block along with logic to implement an increment by one adder. Also required are the 3-bit adder used to compute the current home block of an instruction and the circuitry to implement the WBS suppression state machine and its associated 3-bit comparators that control the write-enable line to the register file.

4.3 Compiler Support

WBS support has been incorporated into the IMPACT-I compiler. The compiler support consists of modifications to the superblock scheduler and the register allocator. The scheduler is implemented in two phases: pre-pass scheduling (prior to register allocation) and post-pass scheduling.

4.3.1 Superblock Scheduler Extensions

There are two major steps for superblock scheduling, dependence graph construction and list scheduling. After the dependence graph is built, a check instruction is inserted into each basic block of a superblock (excluding the first). Control dependences are added from the branches surrounding the check instruction to prevent it from being moved outside of its home block. Next, dependences are added between the check instruction and any PEIs that belong to the same home block as the check. Finally, dependences are added from the check instruction downward to any PEI exactly K home blocks after the home block of the check instruction, where K is the maximal speculation distance defined in Section 4.2. This is to guarantee that no instruction will be speculated more than K blocks away from its home block. These dependences ensure that after scheduling any of these PEIs will be located after the check instruction.

The list scheduler will eliminate a check instruction if no PEIs are actually speculated out of its home block. This is accomplished by removing the check's input dependences as its source instructions are scheduled. If the none of the input dependences for a check have been removed when the branch prior to the check instruction is scheduled, the check can be deleted.

Figure 4.1 shows that I_{12} was added by the scheduler to check for an exception caused by I_6 or I_4. The check instructions are provided a slightly higher priority to push them earlier in their home block to minimize the number of instructions re-executed during recovery.

4.3.2 Register Allocator Extensions

Register allocation in our compiler is done using a global graph coloring approach [19]. The register allocator assumes that all allocatable operands reside within virtual registers. For each of these virtual registers it constructs a live range which consists of the set of instructions where the operand is live. Allocation then proceeds by coloring the interference graph constructed from these live ranges. This basic register allocator has been modified to ensure that the resulting allocation will allow proper recovery in the event of an exception. The modifications prevent the register allocator from destroying the source operands of an SPEI, which can occur if the register allocator reuses a register allocated to a source operand of an SPEI for the destination operand of an instruction from a home block above that of the SPEI.[2]

To ensure a legal allocation, the live range of a source operand of an SPEI must be extended so that the constructed interference graph will prevent the above situation from occurring. The live range is extended by using information provided by the scheduler. The scheduler annotates

[2] Note that speculative instruction I_1 from a home block below that of an SPEI I_2 can reuse the source register of the SPEI. The write-back suppress hardware will automatically prohibit I_2 from destroying the source register of I_1 in case I_2 caused an exception. This is the major advantage of WBS over sentinel scheduling in terms of register pressure.

each SPEI with the last instruction from a home block that is above the SPEI's home block. The register allocator adds to the source operand live ranges, instructions that lie between the SPEI and the instruction indicated by the scheduler and that originate from home block above the SPEI. For example, consider the code sequence before register allocation shown in Figure 4.1. Instruction I_6 is an SPEI, from home block 1. Its source operands, v4 and v7, must not be modified by any instruction from home block < 1. The scheduler indicates that the last instruction with a destination from a home block < 1 is instruction I_2. Thus the register allocator adds instructions I_1 and I_2 to the live range of each source operand of instruction I_6, since they both have home blocks < 1. Adding these two instructions to the live range will prevent the destinations of instructions I_1 and I_2 from being allocated to the same physical register as v4 and v7, while allowing the destination of instruction I_6 or I_7 to do so. Figure 4.2 contains the same code sequence after register allocation. Note that the destinations of instructions I_6 and I_7 were both allocated to physical register r1, while the destinations of instructions I_1 and I_2 were allocated to different registers. In the event of an exception, WBS will ensure correct recovery.

4.4 Write-back Suppression and Context Switching

Unlike subroutine calls, the compiler is not able to insert code to save and restore the processor state during a context switch. Thus, the operating system must save and restore the processor state. Currently, all compiler controlled speculation techniques have additional state to be saved to ensure correct recovery after a context switch. Only restricted speculation that does not permit speculation of PEIs and general speculation that does not recover from exceptions of PEIs do not have increased state. Instruction boosting requires the saving of each additional shadow register file to ensure correct recovery. Sentinel scheduling requires the saving of the exception bits stored in the register file upon a context switch. In addition, the register pressure introduced by the sentinel scheduling to protect the source operands requires an architecture with a larger register file that must also be saved.

WBS adds additional information to the processor state including the current_block register, the current state in the WBS state machine and the valid entries in the push-down stack. All of this information must be protected in the event of a context switch. However, WBS could rebuild its state after a context switch since no registers have been corrupted. By maintaining only the oldest excepting PC, the processor could enter a special *REBUILD* state after returning from a context switch. During this state, the PC would be set to the oldest excepting PC and no updates to the register file would be permitted. Then each instruction could be re-executed until the PC of the instruction where the context switch occurred is reached. At this point, the WBS state machine, the current_block register and the push-down stack would be valid and processing could begin. Since the number of re-executed instructions is not fixed, it would be more efficient to save the the additional state information. However, this option poses an interesting approach to solving the binary compatibility problem.

4.5 Experimental Evaluation

In this section, the effectiveness of WBS is analyzed for a set of non-numeric benchmarks. The performance of WBS is compared with restricted speculation and general speculation.

4.5.1 Methodology

Compiler support for WBS has been implemented in the IMPACT-I C compiler. The IMPACT-I compiler is a prototype optimizing compiler designed to generate efficient code for VLIW and superscalar processors [7]. The benchmarks used in this study are the 14 non-numeric programs shown in Table 3.1. The benchmarks consist of 5 non-numeric programs from the SPECint92 suite and 9 other commonly used non-numeric programs.

The processor model used in this study is a in-order issue superscalar processor with register interlocking. The processor is assumed to have uniform functional units, 1 branch delay slot, and the instruction set of the HP PA-RISC processor. The instruction latencies assumed are those of the HP PA-RISC 7100 (see Table 3.2). The processor is assumed to trap on exceptions for memory load, memory store, and all floating point instructions.

For each machine configuration, the program execution time, assuming 100% cache hit rate is derived using execution-driven simulation. For the experiments the issue rate of the processor is varied from 1 to 8 and the number of integer and floating point registers from 32 to 64.

4.5.2 Results

4.5.2.1 Selection of the Maximum Speculation Distance

The general speculation model was used to schedule all of the benchmarks for a 4-issue and 8-issue processor using a register file size of 64 integer registers, and 64 floating point registers. Each instruction was then tagged with the number of branches that it was speculated above. The benchmarks were traced to get actual execution frequencies for all instructions. The weighted execution frequency of each SPEI was then used to build the graph in Figure 4.5. Considering that a speculation limit of 2^k-1 branches is possible given the addition of k bits to

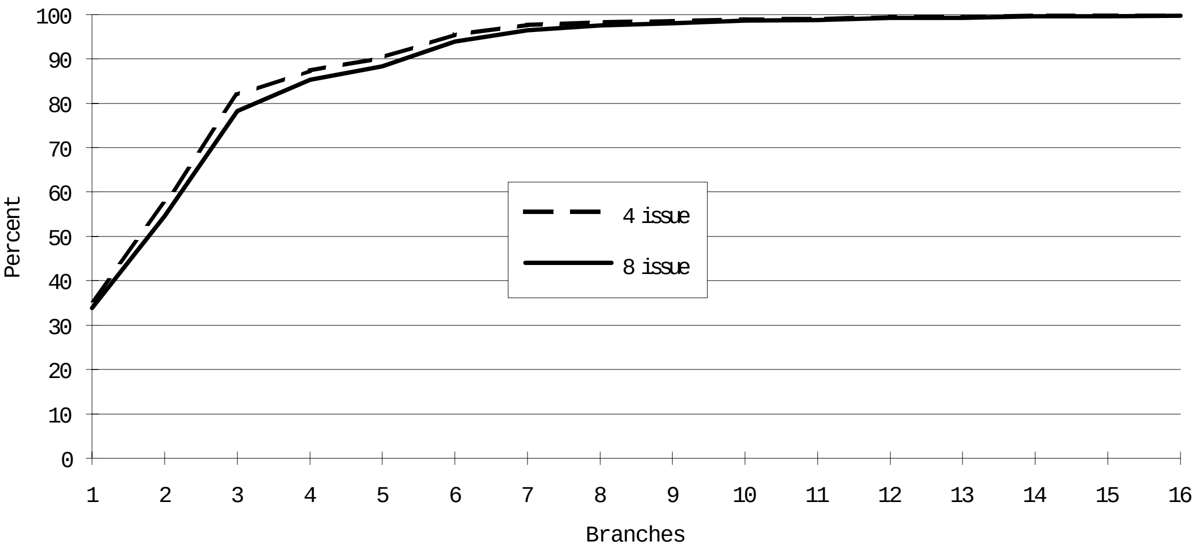


Figure 4.5 Weighted speculation distances for PEIs.

the instruction opcode, viable speculation distances are 3, 7, 15 and 31. A speculation limit of 7 branches was chosen for subsequent experiments since it permitted speculation of 97.7 percent of the PEIs with a 4-issue architecture and 96.4 percent of the PEIs with an 8-issue architecture. The addition of one more bit in the opcode would only increase the percentage of speculated instructions by 2 percent for the 4-issue architecture and 3 percent for the 8-issue architecture but would double the push-down stack size requirements.

4.5.2.2 Comparison of Write-back Suppression and Restricted Speculation

The performance of the WBS scheduling model and the restricted speculation model is compared in Figure 4.6. The base configuration for speedup calculations in this graph is a single issue processor with restricted speculation code scheduling. In general, this figure shows that there are many opportunities for increased ILP using compiler controlled speculative execution techniques.

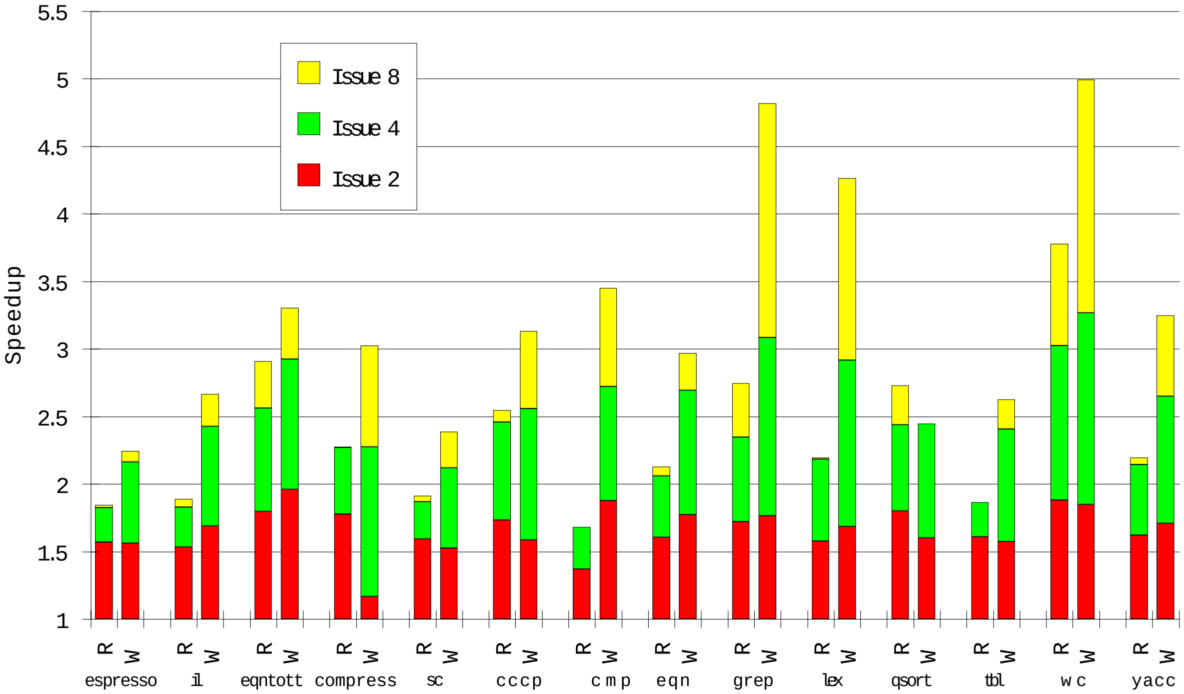


Figure 4.6 Performance comparison of write-back suppression scheduling (W) and restricted speculation (R) with 32 integer and 32 floating-point registers.

There are several observations that one can make on Figure 4.6. First, the additional check instructions required to ensure correct recovery for WBS scheduling for 2-issue result in lower speedups with compress, sc, cccp and tbl than was accomplished by restricted speculation. Second, WBS scheduling provides large performance improvements over restricted speculation for 4-issue and 8-issue processors. The largest speedups are achieved with cmp, grep, lex, and qsort. Finally, qsort is actually slower for WBS scheduling than restricted speculation for an 8-issue processor due to increased register pressure caused by speculation.

4.5.2.3 Comparison of Write-back Suppression and General Speculation

The performance results of the WBS scheduling model and general speculation model are shown in Figures 4.7 through 4.10. All numbers are shown as a percentage of the the performance, defined as one over cycle count, of general speculation. The figures show a general improvement in performance from the single issue to the eight issue architectures. This is expected because the check instructions inserted for WBS have fewer free slots available with lower issue rates and can be better hidden by unused resources for higher issue rates. In Figure 4.10, one exception to the trend is *grep* which is the most parallel benchmark in the experiments. Even for high issue processors, the additional check instructions can increase the number of issue slots required and therefore increases the cycle count.

The second trend to note is that WBS shows about the same (within 2 percent) or slightly higher performance with 64 registers than with 32 registers. This indicates that register pressure introduced by WBS is small and does not require a larger register file to attain comparable performance to general speculation. This differs from sentinel scheduling which experiments show requires a larger register file size to achieve comparable performance to WBS scheduling [35].

One notable exception to the trend discussed in the previous paragraph is *compress* shown in Figure 4.9 which shows a decrease in the performance from 32 to 64 registers. The schedule generated for *compress* using general speculation suffers from a great deal of register pressure with only 32 integer registers.

Overall, WBS achieves between 86 and 100 percent of the performance of general speculation. These numbers were demonstrated for processor issue rates varying from 1 to 8 with register file sizes of 32 and 64. The experiments show that the exception handling required for WBS introduces little performance overhead while ensuring accurate exception recovery with compiler controlled speculative execution.

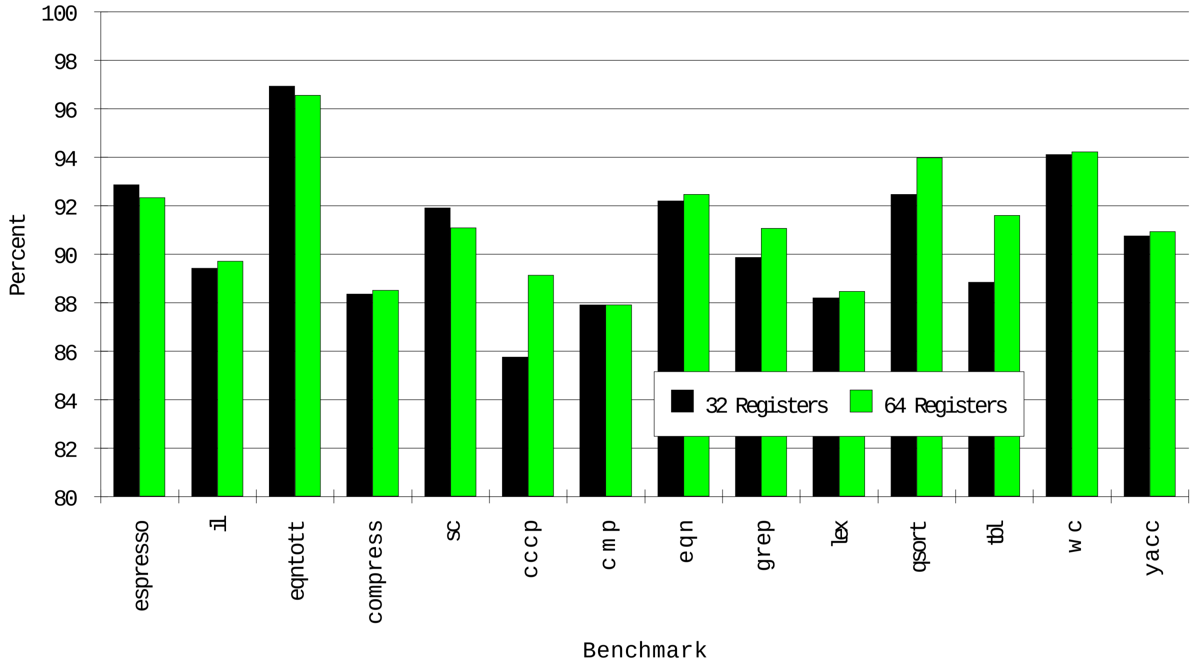


Figure 4.7 Performance results of write-back suppression relative to general speculation using an issue 1 processor.

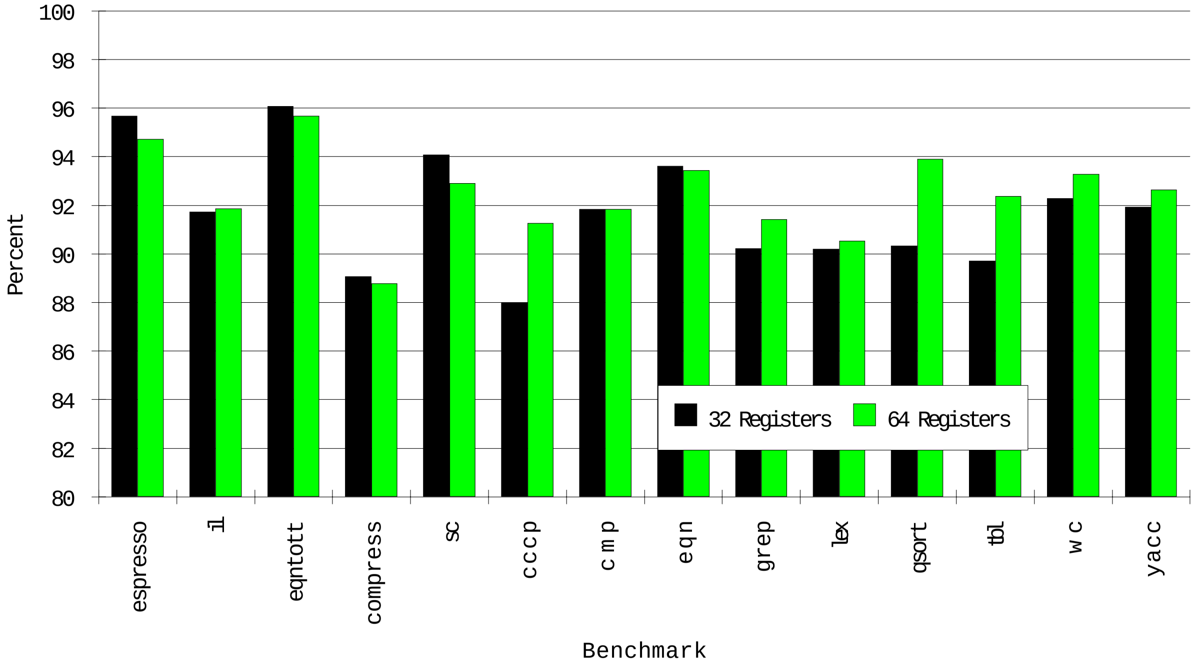


Figure 4.8 Performance results of write-back suppression relative to general speculation using an issue 2 processor.

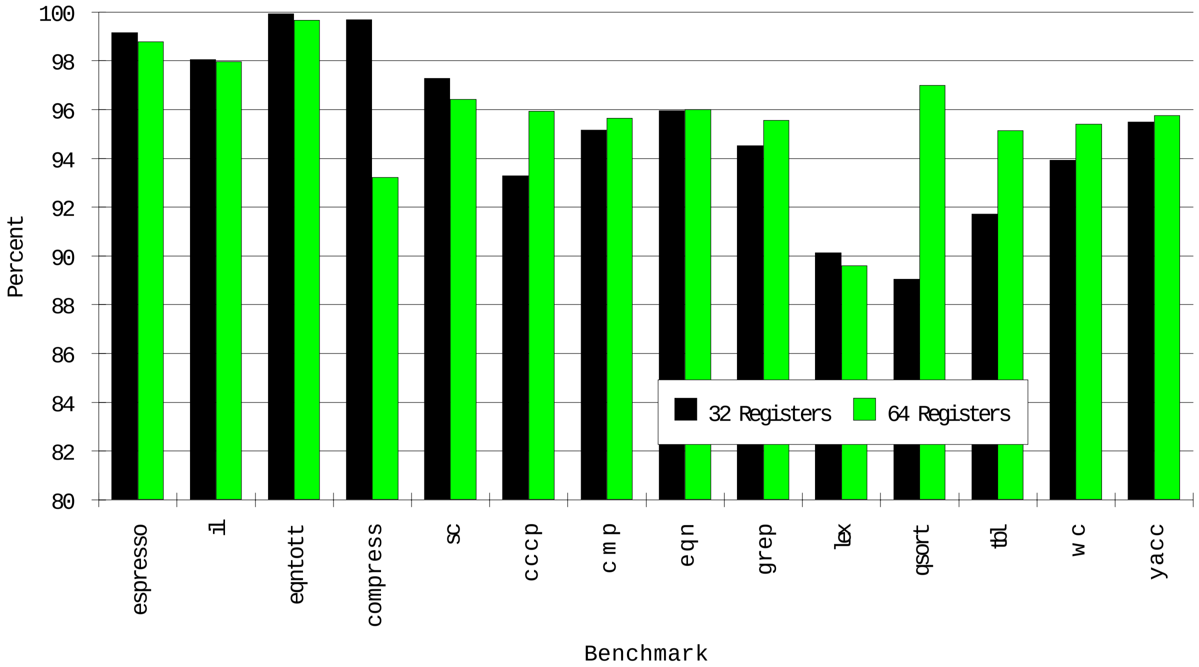


Figure 4.9 Performance results of write-back suppression relative to general speculation using an issue 4 processor.

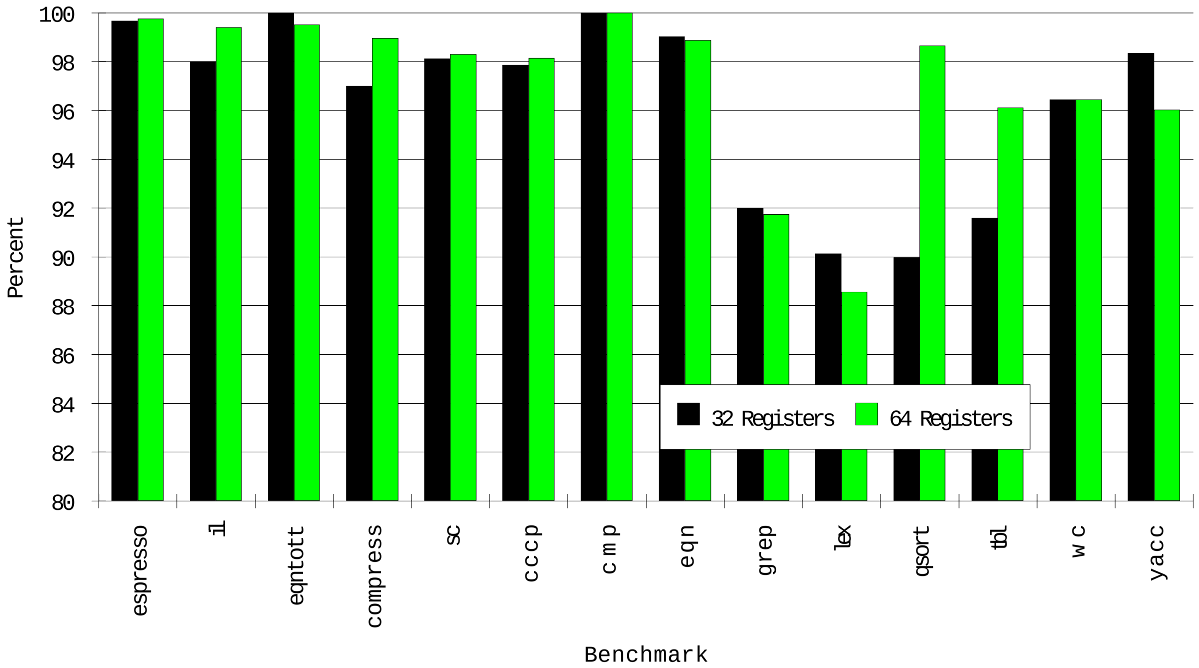


Figure 4.10 Performance results of write-back suppression relative to general speculation using an issue 8 processor.

4.6 Conclusion

This chapter has introduced a new architectural scheme referred to as *write-back suppression.* This scheme systematically suppresses register file updates for subsequent speculative instructions. We have shown that with a modest amount of hardware, WBS supports accurate reporting and handling of exceptions for compiler-controlled speculative execution with minimal additional register pressure.

Experiments using a prototype compiler implementation and hardware simulation indicate that ensuring accurate handling of exceptions with WBS incurs very little run-time performance overhead. In particular, experimental results from a series of non-numeric benchmarks indicate that WBS can achieve from 86 to 100 percent of the performance gains of general speculation scheduling and still ensure correct exception handling in all conditions.

CHAPTER 5

SAFE SPECULATION

5.1 Overview

This section introduces intra-procedural and inter-procedural analysis techniques that are utilized to identify potentially excepting instructions that will not introduce run-time errors and are thus safe to speculate. As such, this analysis is used to extend the scheduling freedom available for restricted speculation in the *avoid errors* category. The analysis algorithms discussed in this chapter are also used to identify instructions that prevent operands from reaching potentially excepting instructions that will cause them to except. From this point on, usage of these analysis techniques will be applied to a speculation model called *safe speculation.*

There are three benefits that result from identifying potentially excepting instructions that are always safe. The obvious benefit is that it allows speculation of an instruction to hide long instruction latencies. The second benefit becomes visible after applying transformations such as loop unrolling to increase the available ILP for wide issue processors. Unsafe instructions in the middle of these loops can limit or prevent overlap of the unrolled loop iterations, thus losing most of the benefits of unrolling. By proving that these instructions are safe, the scheduler will be able to overlap of these iterations.

The third benefit comes from identifying potentially excepting instructions that are not executed on all paths of the program. Kennedy's work on safety of code motion uses knowledge of safety to permit loop invariant code removal of conditionally executed potentially excepting instructions [42]. If this instructions are not identified as safe, then this optimization could introduce a run-time error.

The safety analysis discussed in Sections 5.1.1 through 5.1.3 are performed prior to loop optimizations, superblock formation and superblock optimizations. This permits the analysis information to be used when performing these optimizations. Additionally, it is possible that some optimizations may invalidate the analysis. As such, constraints can be imposed to prevent this invalidation.

5.1.1 Determining Safety through Operand Inspection

It is possible to determine that some potentially excepting instructions are safe by trivial inspection of the explicit operands. One simple example is a divide or remainder whose denominator is a non-zero constant. In this case, the divide and remainder are always safe to speculate. While these cases are obviously safe, the benefits from them are dependent upon the execution time of the divide due to the strength reduction optimization [43]. Strength reduction is used to convert a divide by constant integer into a series of shifts and subtracts. Thus, the divide can be replaced by a series of instructions that are always safe.

The other class of instructions that can be determined safe by operand inspection are loads. These references can be broken up into two groups. The first group includes loads that are based upon a label plus an immediate offset. By querying the symbol table, it is possible to determine whether the offset is within the bounds of the variable and thus ensure safety of the load.

The second group are loads are inserted by the compiler while generating the intermediate representation for a function and as a result of register allocation. These loads can be proven to be safe by ensuring that their constant offsets will be within the bounds of the function stack. Examples of loads that meet this criteria are:

(1) Loads used to pass parameters between function calls.

(2) Loads used for spill code.

(3) Loads used to transfer data between integer and floating point register files [30, 31, 44].

(4) Loads used to load constants into floating point registers [30].

5.1.2 Determining Safety through Equivalent Instructions

Kennedy proposed an algorithm that can ensure that the insertion of an expression is safe before a conditional branch if all paths that reach the expression in question are reached by an identical expression [42]. This rule ignores the values of operands and requires reaching expression tests [41]. The advantage of this test is it permits speculation of not only load instructions but also divide, remainder and floating point instructions. The benefits of this test are somewhat limited since global optimization techniques will typically eliminate redundant computations. However, global optimizations are not able to perform redundant load elimination if an ambiguous store is available on a path that reaches the load [41]. Thus, there are some potential advantages of this safety analysis in the case of load instructions.

If all paths that reach an integer divide or remainder pass through another divide or remainder with the same denominator, the earlier divides or remainders will except if there is a potential for an error. Thus, the subsequent divide or remainder are safe to speculate as long as they are not speculated above the earlier instructions. These earlier instructions will be called sentry instructions.

5.1.3 Determining Safety through Graph Traversal

Going back to the basic definition of safe speculation requires that all possible values that reach the operands for a potentially excepting instruction must not cause an exception. If the values for an operand along any path can not be determined, then it is possible that the potentially excepting instruction could cause an exception. For example, the operand values that reach the denominator of a divide instruction must be non-zero from all paths. If this is not possible, then the divide is unsafe. Figure 5.1 contains a divide instruction in block D with the divisor (*register r14*). There are four paths which reach the divisor. The first two paths are from blocks A and B where the divisor is initialized to 7 and 23 respectively. The third path traverses CEFCD where there are no definitions of the divisor. Also, the fourth path, CDFCD, does not redefine the divisor. Thus, the denominator of the divide is reached by non-zero values along all paths which ensures that the divide is safe to speculate. In this particular case, not only is the divide safe to speculate, but it is also invariant which permits loop invariant code removal.

Figure 5.1 can be used to show the required steps to determine that an invariant load in a loop is safe or a load that is not in a loop is safe. In this case, the base address of the load found in block F is not visible within the scope of the loop. However, the base operand, *register r2*, is defined in blocks A and B. Using the two labels that reach this load, it is possible to determine from the symbol table whether an offset of 4 is legal for both structures. If it is, then the load is always safe to speculate into block C.

Large portions of program execution time is spent in loops. As such, it is not only necessary determine that invariant references are safe but also determine that variant references are safe. If a loop can be characterized as a *for-loop* that it is possible to determine the number of

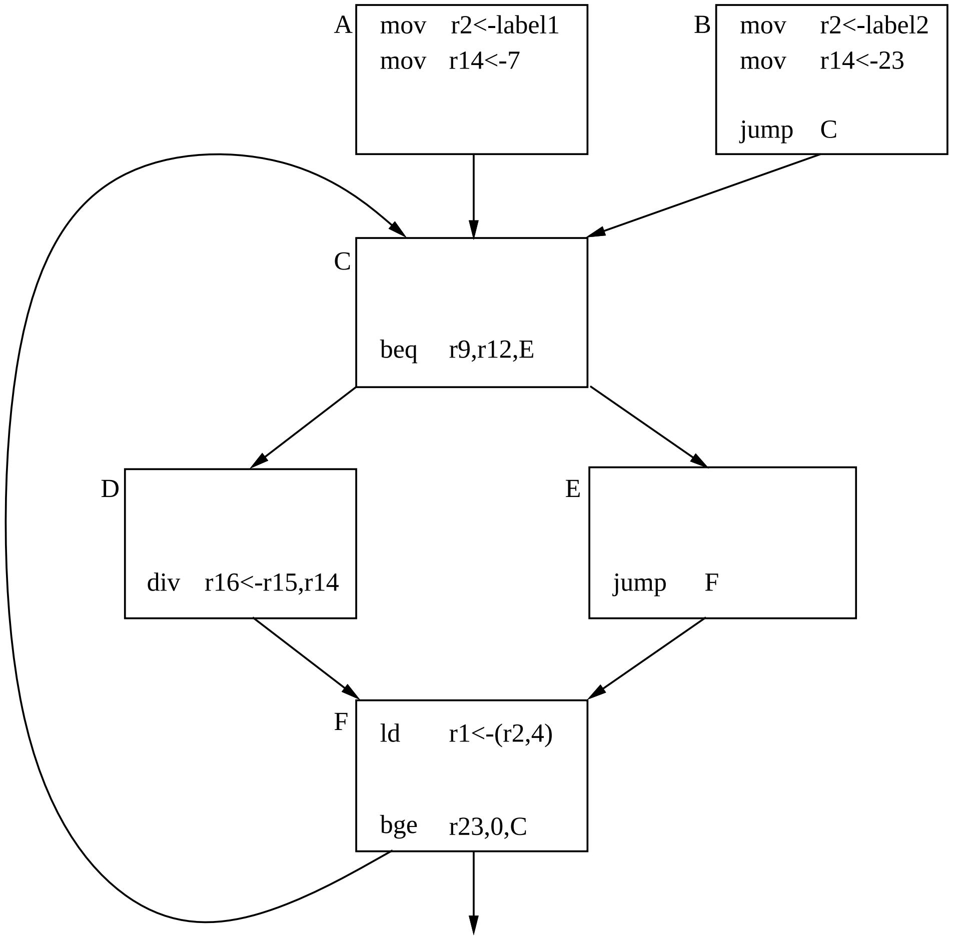


Figure 5.1 Example where potentially excepting instructions are always safe.

iterations that a loop will span. Additionally, if a load can be determined to be an array-type load, knowledge of the loop span and the array reference pattern, can be used to determine that these load accesses are safe. Figure 5.2 shows a case where the load from block D in the loop is variant. For variant loads, it is not only necessary to determine the base address of the load, but it is also necessary to determine the worst case trip count for the loop in order to determine the load reference pattern. Thus, it is necessary to ensure that the load is safe on all iterations of the loop. This loop is complicated by the jsr in block C because some iterations, will produce a new base address for the load. Therefore, to determine that this load is safe, it is also necessary to perform inter-procedural analysis to determine the return value from the function and then verify that the loop bounds will stay within the bounds of that variable as well as the variable provided from block A. This particular problem occurs in HP and SUN implementations of the C function *getc* which is typically implemented as a macro call. The resultant C code will test to see if the buffer is empty. If the buffer is empty, a call to *filbuf* is performed to refill the buffer. Therefore, this example also motivates the need for inter-procedural analysis.

These examples have shown the requirements that must be followed to ensure that a potentially excepting load is safe along all paths. If all of the relevant steps are not possible, then the load must be considered unsafe for speculation.

(1) Identify all possible variable addresses that reach the load.

(2) Identify all possible initial offsets that reach the load.

(3) If the load is variant within a loop, identify the iteration space for the loop including start value, step and end value. This information is used to determine the possible span of addresses that the load may reference inside the loop.

(4) Verify with the symbol table that all possible load references are legal.

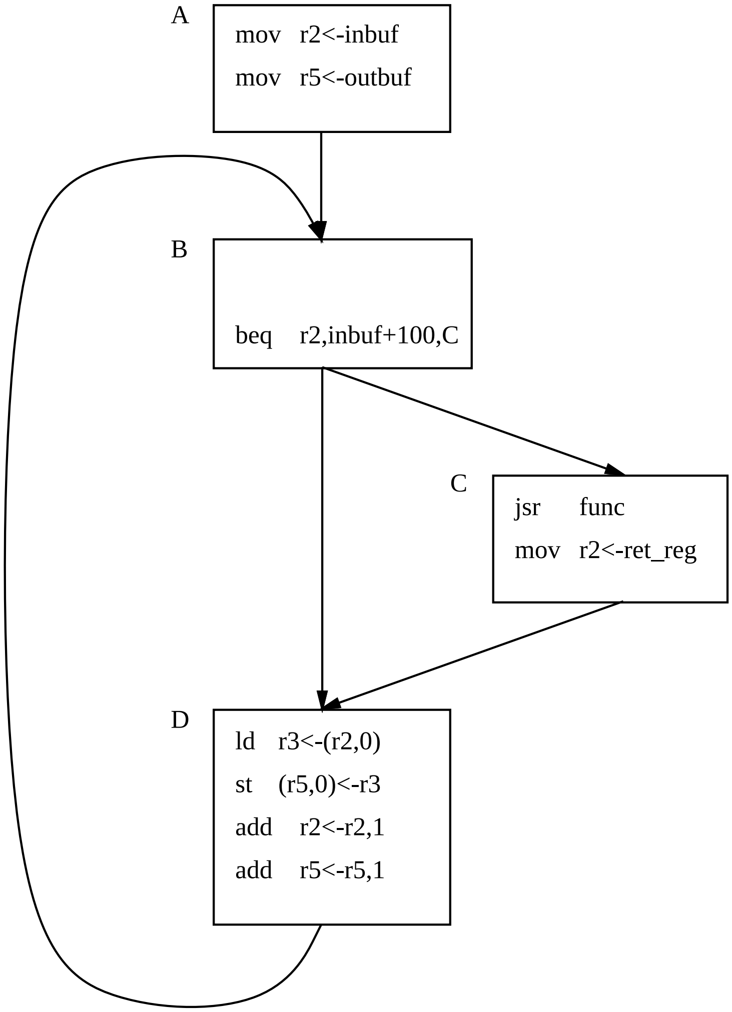


Figure 5.2 Example where the number of loop iterations and inter-procedural analysis is required to determine safety.

5.1.4 Ensuring Safety in the Presence of Optimizations

Conditional branches can be used in the graph traversal processes to prevent unsafe operands from reaching a potentially excepting instruction. While the absence of this information can be used to ensure that a potentially excepting instruction is safe, the missing information about the branch could result in illegal scheduling later after superblock formation. Figure 5.3 can be used to illustrate this possibility. The divisor (*register 14*) in block D is safe along the path through block A. Additionally, the conditional branch at the end of block B prevents a zero from reaching the divisor. Thus, the load is safe along all paths. Since the divide is also invariant, than loop invariant code removal could move the divide to the loop pre-header. Superblock formation could then merge the preheader into the same superblock as block B. Without maintaining the knowledge that the divide was conditionally safe along the path leading to the branch, later speculation could schedule the divide above the branch which could result in run-time errors. Thus, it is necessary to maintain dependence information about the branch to prevent this error.

A memory reference to the safe address also functions as a sentry instruction. However, natural memory dependences are sufficient to prevent re-ordering of references which could lead to run-time errors. It is important to note that the current implementation of safety analysis is performed after redundant load elimination [41]. Therefore, an earlier memory reference may only be a store to the same address or load that is separated from the potentially excepting load by a store to an unknown address. In either of these cases, re-ordering the load can not be speculated above the reference to the same address. For the rest of the dissertation, conditional branches and memory references that are used to ensure safety will be called *sentry* instructions.

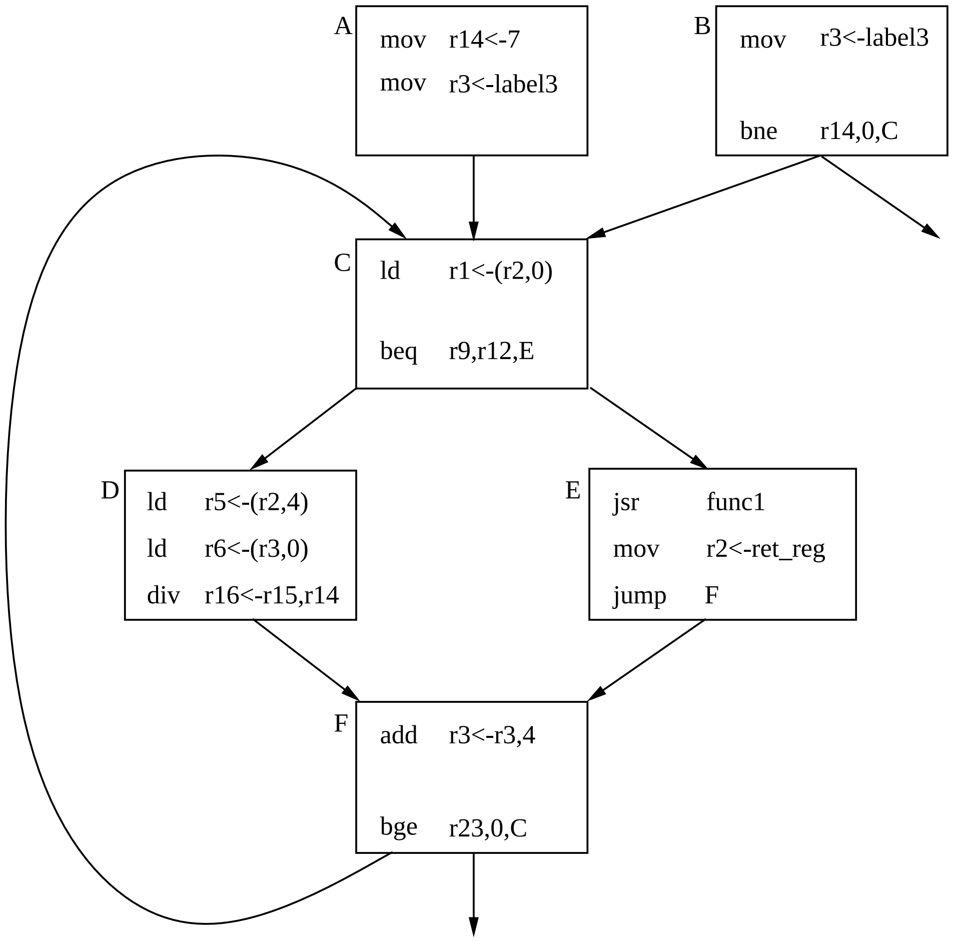


Figure 5.3 Examples where sentry instructions prevent run-time errors.

One final requirement of the safety analysis is to maintain safety in the presence of subsequent optimization passes. Since the analysis that determined the span of load references was based upon the original loop, the safety of load references is constrained to the original patterns determined by the original worst-case trip count. Unfortunately, if the loop contains very little ILP, this analysis may provide insufficient opportunities for speculation. Thus, the analysis has provided very little benefit. In order to permit greater ILP in the presence of these optimizations, it is necessary for the analysis to not only identify safety within the current loop iteration but across loop iterations. Unfortunately, this opens the opportunity for ILP optimizations to violate the safety constraints. Figure 5.4a illustrates a simple loop prior to unrolling and ILP optimizations. Using profile information, the loop unrolling optimization decides to unroll the loop 3 times. Figure 5.4b illustrates the loop after unrolling, induction variable expansion and accumulator variable expansion.

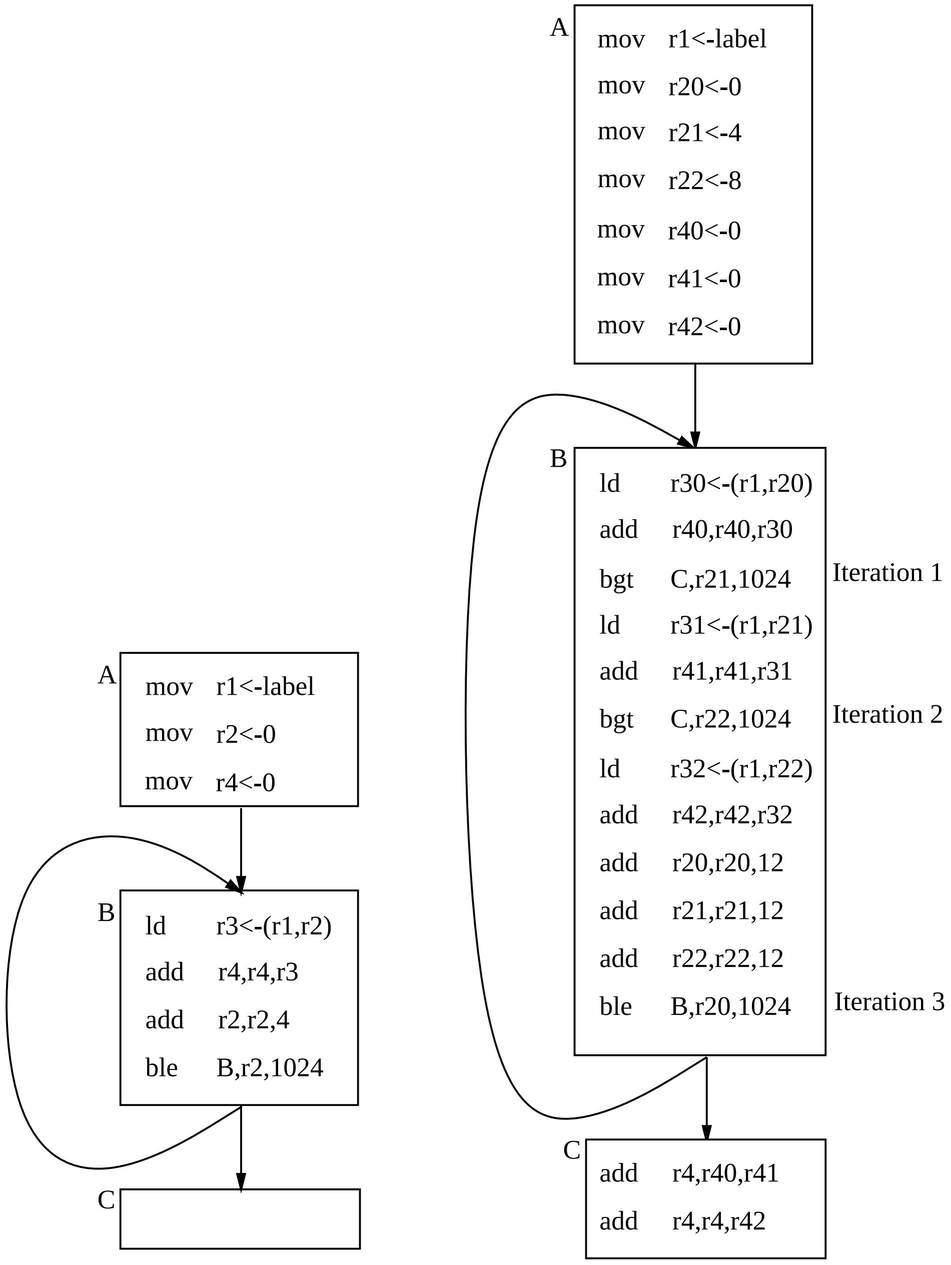
The potential problem arises because loop unrolling was based upon profile information to determine the average number of loop iterations. To provide the maximum overlap of loop iterations, it is desirable to provide unlimited speculation of the loads from each of the unrolled iterations. As such, it is possible to execute 2 additional memory accesses beyond the bounds of the buffer starting at label. Assuming that label is a pointer to a static array containing 256 entries, this would result in 258 references. Since, the analysis process determined that the original loop was going to access only entries within the original 256 entry array, these additional references are caused by the speculation only and not by a programming error. As long as there are at least two words of data located after the array, the additional references will not cause a run-time error.

However, consider the case where the buffer is not a statically allocated array but a dynamically allocated array of 256 entries. In this case, there is no method for the compiler to guarantee at compile-time that the references will not cause an access violation. As such, it is necessary for the analysis process to constrain unrolling to a multiple of the size of the array. This will ensure that the memory reference pattern stays within the legal bounds of the dynamically allocated array. Unfortunately, this could restrict the optimization phases, which could result in poorer performance.

5.1.5 Related Work

Kennedy proposed an algorithm that assumes definition free paths reaching a subsequent instruction [42]. The focus of this work was to identify conditionally executed instructions within loops that will not cause exceptions. This information is critical when performing invariant code removal to prevent program termination. The opportunities that this safety condition can be applied to are limited after optimization phases due to optimizations such as redundant load elimination.

Bernstein and Rodeh, proposed an algorithm that determines the safety of load instructions [45]. Two classes of safety are discussed. The first class requires that there is a memory reference to the exact same address of a load that is beneficial to speculate. If this condition is met, then these earlier memory references act as sentry instructions for the load. This technique is similar to the technique proposed by Kennedy. The second class of safety does not require a reference to the exact same address. Starting with a memory reference at some base address with a constant offset, it analyzes the address computations between the base memory reference to a subsequent load whose address is based on the original address. If the difference between



```
A
mov   r1<-label
mov   r2<-0
mov   r4<-0

B
ld    r3<-(r1,r2)
add   r4,r4,r3
add   r2,r2,4
ble   B,r2,1024

C
```

```
A
mov   r1<-label
mov   r20<-0
mov   r21<-4
mov   r22<-8
mov   r40<-0
mov   r41<-0
mov   r42<-0

B
ld    r30<-(r1,r20)
add   r40,r40,r30
bgt   C,r21,1024        Iteration 1
ld    r31<-(r1,r21)
add   r41,r41,r31
bgt   C,r22,1024        Iteration 2
ld    r32<-(r1,r22)
add   r42,r42,r32
add   r20,r20,12
add   r21,r21,12
add   r22,r22,12
ble   B,r20,1024        Iteration 3

C
add   r4,r40,r41
add   r4,r4,r42
```

a) Original loop

b) Unrolled loop after ILP Optimizations

Figure 5.4 Example indicating the need to control optimization to maintain safety.

the original memory reference and the final memory reference can be shown to be a fixed size k which is the size of a page in the memory system, the load can be made safe. The end of the data segment will then be padded by an additional page of memory to ensure that the load will not reference outside the legal data segment and suffer the risk of an access violation. Instead of proving that a reference is safe, this approach ensures that the load will not cause an exception by padding the end of the data segment. This algorithm is limited by the presence of loops that may iterate over large arrays as commonly found in scientific applications. It can only allow speculation of loads across iterations up to the point that the difference between the original base address and the subsequent load addresses are within the bounds of one page of memory. This technique can be extended to increase the padding distance, but this could require padding comparable to the largest data structure referenced within a loop. Additionally, this approach does not work for dynamically allocated data structures.

Mahadevan and Ramakrishnan present a region scheduling paper that utilizes some of the techniques similar to Bernstein's work to prevent exceptions from occurring [46]. Additionally, they discuss techniques that can be used to prevent array referenced loads from excepting in the presence of loop optimizations. Their rules for safe loads are similar to those proposed in this research. However, their analysis techniques are performed over a region which is composed of a set of basic blocks comparable to a superblock or a trace. Thus, their analysis focuses on intra-procedural analysis. Their techniques are dependent upon data structure size information, to determine the safety of a load. The loop information is used to make decisions on padding arrays to prevent access violations. Since global optimizations tend to move instructions from frequently executed paths to less frequently executed paths, it is possible for the base address to be unavailable within the region being scheduled and thus force the load to be considered unsafe. The intra-procedural and inter-procedural analysis proposed in this dissertation has a greater scope of visibility and thus can resolve this problem. Additionally, safety analysis presented in this dissertation is performed earlier in the compilation phase to benefit not only scheduling but also optimization phases.

5.2 Compiler Support

This section describes the algorithms used to perform intra-procedural and inter-procedural analysis to identify potentially excepting instructions that are safe for optimization and scheduling phases. Additionally, it describes the algorithms used to build C program call-graphs required to perform the inter-procedural analysis. Sections 5.2.1 and 5.2.2 describe the algorithms used to evaluate all possible values that reach an operand in an instruction. These algorithms form the basis for identifying safe potentially excepting instructions, evaluating loop iteration space, and identifying function call-sites to build the program call-graph. Section 5.2.3 describes the algorithms used to build the program call-graph. Section 5.2.4 describes the algorithms that are used identify side-effect free subroutine calls. Finally, Section 5.2.5 describes the algorithms that are used to identify safe potentially excepting instructions for each of the categories described in Sections 5.1.1 through 5.1.3.

5.2.1 Generating Use-Def Graphs

A use-def graph is a cyclic or acyclic graph G composed of nodes V and edges E. Each node represents a basic block in a function that is reached with a register file and memory state. Each edge represents a path in a function that is reachable between two nodes. A basic block may be represented by multiple nodes indicating that it is reached along multiple edges in the graph with different states. Each node contains 0 or more instructions from the basic block that contribute to the state definition along a specific path. In the worst case, the graph will enumerate all paths through a function. The objective of these algorithms is to build a use-def graph containing the minimum number of nodes and instructions within each nodes as well as edges that still accurately represents the program state required to define the source operands of a specified instruction. In the acyclic case the instruction represents the single root in the graph. The graph then has the appearance of an inverted tree with the leaves functioning as the start points in the graph.

Figure 5.5 contains the main algorithm used to build use-def graphs. The algorithm is provided an instruction in a function that will be the initial search point for the graph. Additionally, the algorithm is provide a set of initial search criteria that can be in the form of instruction source operands or memory addresses. The algorithm begins by creating a new use-def graph, creating the initial node in the graph with the starting instruction and initializing the search sets. There are two search sets in the algorithm. The first search set contains all source operands that are unknown along a given path. The second search set contains all load addresses of the form *(base + offset)* that need to be resolved during the traversal process. The algorithm **Recursively_Build_Use_Def_Graph** is used to generate the use-def graph. The last two steps of use-def graph generation are required to make the use-def graph evaluation phase more efficient.

Figure 5.6 presents the recursive depth first traversal algorithm used to build the use-def graph. First, the current basic block is searched for instructions that produce results that are in either of the search sets. This will be discussed in more detail later. If there are any registers or memory addresses in the current search sets, then Traverse_Basic_Block was unable to resolve

Algorithm Build_Use_Def_Graph: Given a start point in a function, and operands required for search, build the use-def graph that produces the source operands for the starting instruction.

```
initialize new use-def graph

create the initial node in the graph containing the starting instruction

initialize search sets based upon start instruction

Recursively_Build_Use_Def_Graph starting at the start point with the initial
    search sets

Identify_Cycles in the use-def graph

for each start_node in use-def graph
{
    start_node->level = 0
    Levelize_Use_Def_Graph ( start_node, 1 )
}
```

Figure 5.5 The Build_Use_Def_Graph algorithm generates a levelized use-def graph with cycles identified.

all unknown source operands, thus it is necessary to continue the traversal process with the source control blocks.

Next, the algorithm determines if a node in the graph exists for the current source basic block. If not, it creates a node and sets the initial search sets for the node to the current search sets. Next it will recursively call the Recursively_Build_Use_Def_Graph to continue the graph building process. If one or more nodes in the graph do exist for the current basic block, then each of the nodes initial search sets is compared with the current search set. If an exact match is found, then an arc is added to the existing node in the graph and the search process terminates. Thus, the graph traversal process will terminate when their is no benefit in traversing a path that already exists in the graph. In this heuristic, benefit in traversing a path is based upon whether a path has been traversed with the same set of search criteria before. If the path has not been searched with the same set of search criteria, it is possible that new instructions will

Algorithm Recursively_Build_Use_Def_Graph: Given a start point in a function, recursively build the use-def graph starting at the start instruction until there are no remaining search constraints. The use-def graph is grown from the start point upwards through the each basic block to the source basic blocks.

```
current search sets = Traverse_Basic_Block ( current node, search sets )

if (current search sets are not empty)
{
    for each source arc to current basic block
    {
        if (the source block does not exist in the use-def graph)
        {
            create a new node for source block using the current
                search sets

            Recursively_Build_Use_Def_Graph based upon the current search
                sets.
        }
        else if (the source node not been reached with the state of the
            current search set)
        {
            Recursively_Build_Use_Def_Graph based upon the current search
                sets.
        }

        add an arc from the current node to the source node in the graph
    }
}
```

Figure 5.6 The Recursively_Build_Use_Def_Graph algorithm generates a use-def graph starting at the initial start point.

be added to an existing node. This processes is repeated for each source arc to the current block until a steady state is reached.

Figure 5.7 presents the algorithm used to traverse a basic block in order to add instructions to the use-def graph and update the memory and register search sets. The objective is to add the minimum number of instructions from a basic block but still gather sufficient information as to ensure accurate operand evaluation. In all cases, conservative assumptions will be made when there is any question about whether the instruction should be added. Each instruction in the basic block will be evaluated for addition to the instructions in the current node. Instructions are broken up into five categories based upon the requirements for addition to a node's instruction list as well as the actions performed on the current register and memory search sets based upon adding the instruction.

If the destination register of a load instruction is in the current register search set, then the load is added to the current node's instruction list. The destination register is removed from the current search set and it's source operand registers are added to list. Additionally, the address for the load will be added to the current memory search set since the contents of at that address are what define the destination of the load. Store instructions may be added to the list for two reasons. If the address of the store is in the current memory search set, the the store is added to the list. Also, the store will be added if it's address is ambiguous with any of the addresses in the current memory search set. Thus, any load whose source address is not explicit will cause the addition of all stores reach through the traversal process. Additionally, any store whose target address is not explicit will be added to the list. As with load instructions, the source operands for the store are added to the current register search set.

Subroutine call instructions are added under two conditions. First, if the return parameters of the subroutine are currently in the current register search set, then the subroutine produces a result that is required for the use-def graph evaluation phase. The second condition will conservatively add the instruction to the node list if the subroutine call is not side-effect free and the current memory search set is not empty. Since no inter-procedural analysis is performed during the graph building phase, it is not possible to know whether the subroutine may change the memory state that ultimately affects the value reaching a load. More sophisticated memory disambiguation could be used to reduce this conservative assumption.

Arithmetic instructions are added to the list of instructions for a node if the destination register is in the current register search set. The destination register will then be removed from the set and all register source operands will be added to the list. Finally, all conditional branches are added to the current node's instruction list and their source operands are added to the current register search set. This ensures that all potential branch conditions that may guard against exceptions are visible to the graph evaluation phase.

At any point, if the search sets become empty, then traversal through the block will terminate. For this to occur, an instruction must define a result in the current search set but not introduce new search conditions. Thus, with optimized code, search conditions only terminate upon encountering move instructions with label or constant source operands. The presence of ambiguous load conditions will prevent the memory search set from ever becoming empty.

The use-def graph evaluation only makes two passes down the use-def graph to determine the values that can reach each instruction. The first pass only covers acyclic edges in the graph while the second pass propagates values around cycle edges as well as acyclic edges. Figures 5.8 and 5.9 are used to mark cycles in the graph to support the evaluation phase. These algorithms are based upon the algorithms **DFS** and **DFS_Visit** found in [6].

To ensure accurate evaluation of the graph, it is necessary to delay evaluation of a node until all source nodes have been evaluated. This can simply be done after cycles have been evaluated by not traversing cyclic edges. Figure 5.10 increments a nodes level to the one greater than the lowest level node reaching it. Thus, the node is pushed down the graph. The results of this algorithm produce a result comparable to a topological sort [6]. As a result of the leveling processes all start nodes in the graph will be at level 0.

Algorithm Traverse_Basic_Block: Given a start block in the basic block and the current search sets, add instructions to the use-def graph for the current node that produce results that affect the current search sets.

```
for each instruction in the current basic block
{
    if ((subroutine call instruction) && ((subroutine produces a result) &&
         (the result is in the current search set)) ||
        ((the subroutine is not side-effect free) &&
         (there are any memory references in the current search set)))
    {
        add the instruction to the current node in the use-def graph
        add the source operands to the current search set
        add the parameter registers to the current search set
    }
    else if (conditional branch instruction)
    {
        add the instruction to the current node in the use-def graph
        add the source operands to the current search set
    }
    else if ((load instruction) && (destination in current search set))
    {
        add the instruction to the current node in the use-def graph
        add the source operands to the current search set
        add the load address to the current search set
    }
    else if ((store instruction) &&
             ((store address is in current memory search set) ||
              (store address is ambiguous with any memory address
               in current search set)))
    {
        add the instruction to the current node in the use-def graph
        if (there is an exact match of the store's address in the
            current memory search set)
            remove the store from the current search set
    }
    else if ((instruction produces a result) &&
             (destination in current search set))
    {
        add the source operands to the current search set
        add the instruction to the current node in the use-def graph
    }
    if (current search sets are empty) exit loop
}
```

Figure 5.7 The Traverse_Basic_Block algorithm adds relevant instructions to the current node in the use-def graph.

Algorithm Identify_Cycles: Given a use-def graph with multiple start nodes, identify all cycles. This algorithm is based upon the algorithm **DFS** from [6].

```
for each node in the use-def graph
    node->color = WHITE

for each start_node in the use-def graph
{
    if (node->color == WHITE)
        DFS_Identify_Cycles( start_node )
}
```

Figure 5.8 The Identify_Cycles algorithms is used to identify cycles in the graph to permit correct graph levelization.

Algorithm DFS_Identify_Cycles: Given a node in the use-def graph identify cycles in the graph. This algorithm is based upon the algorithm **DFS_Visit** from [6].

```
node->color = GRAY

for each destination_arc of this node
{
    destination_node = destination_arc->destination_node

    if (destination_node->color == white)
        DFS_Identify_Cycles( destination_node)

    else
        destination_arc->type = CYCLIC
}

node->color = BLACK
```

Figure 5.9 The DFS_Identify_Cycles algorithm is used to identify cycles in the graph to permit correct graph levelization.

Algorithm Levelize_Use_Def_Graph: Given a node in the use-def graph and a current **level** for the node, levelize the graph using a DFS traversal.

```
for each destination_arc of this node
{
    if (destination_arc->type == CYCLIC)
        continue;

    destination_node = destination_arc->destination_node

    if (level > destination_node->level)
    {
        destination_node->level = level

        Levelize_Use_Def_Graph ( destination_node, level + 1 )
    }
}
```

Figure 5.10 The Levelize_Use_Def_Graph algorithm is used to push the level of all nodes to the lowest levels in the graph that reach them.

In an effort to reduce the run-time overhead, two heuristics were added to this Recursively_Build_Use_Def_Graph. The first heuristic prevents traversing too far down a given path. This limit is currently set to 50 blocks for a given enumerated path. The second heuristic prevents traversing through too many blocks in the function. This limit is currently set to 4000 blocks. Currently, only the limits of the first heuristic have resulted in termination of the traversal process. While these heuristics can ultimately affect the amount of useful information found during the graph evaluation process, it has not led to significant performance impact for three reasons. First, ambiguous stores will result in the addition of search criteria that are not always beneficial which can cause traversal beyond useful points. Second, due to loops, we will iterate through the same sequence of paths until a steady state is reached. This traversal process does not always result in additional useful instructions. The third reason actually points out one of the limitations with the current loop analysis. Many of the paths that terminate due to the path length heuristic are in loops that whose iteration spans are unknown. This includes loops that iterate through linked lists and loops that function like state machines such as in parsers.

Figures 5.11 through 5.19 provide an example of these algorithms using the inner-loop of *wc*. Due to the size of the function shown in Figure 5.11, only the inner loop shown in Figure 5.12 will be used in this example [1]. This figure shows the control-flow graph for the inner-loop with the instructions numbers and opcodes in their respective basic blocks. The instructions with their operands are provided in figure 5.13 for reference. The instruction format assumes that the destination of the instruction is located immediately after the opcode. In this example, we will build the use-def graph required to determine if instruction 42 is safe.

The initialization process produces the graph and state information shown in Figure 5.14. The initial register search state is set to contain register 21 which is the only unknown source operand in op 42. The initial memory search state is empty. Even though op 42 is a load, the only requirement for safety is to determine that the address of the load will not cause an exception. Therefore, it is not necessary to know the contents of the load at this point. Instruction 42 is placed into the instruction list for the node representing cb 9. Finally, the current register and memory search states are initialized to the same state as the initial search state of cb 9. Recursive_Build_Use_Def_Graph will then be called to process cb 9.

Figure 5.15 show the configuration of the use-def graph after traversing through all instructions in cb 9. Instruction 38 is added to the instruction list of the current node since it defines register 21 which is in the current register search set. Register 21 is removed from the current search set and the only unknown operand, register 3 is added. The memory address, (r3 +

[1] In actuality, the graph building process traverses through the inner-loop, the outer-loop and upwards to the first block in the function.

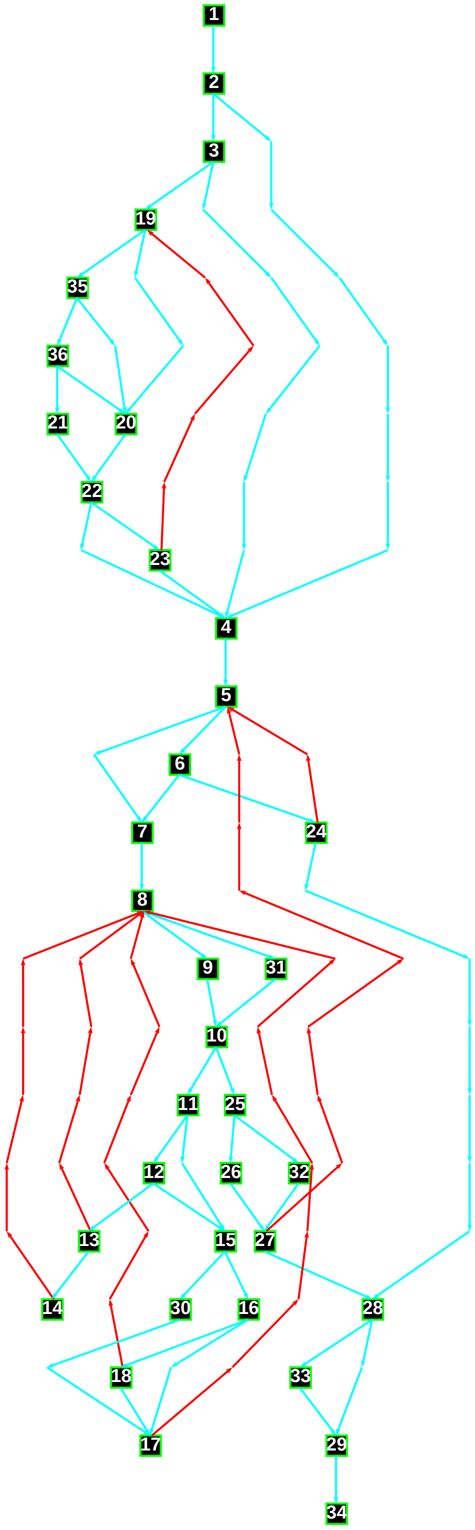


Figure 5.11 Main function for wc.

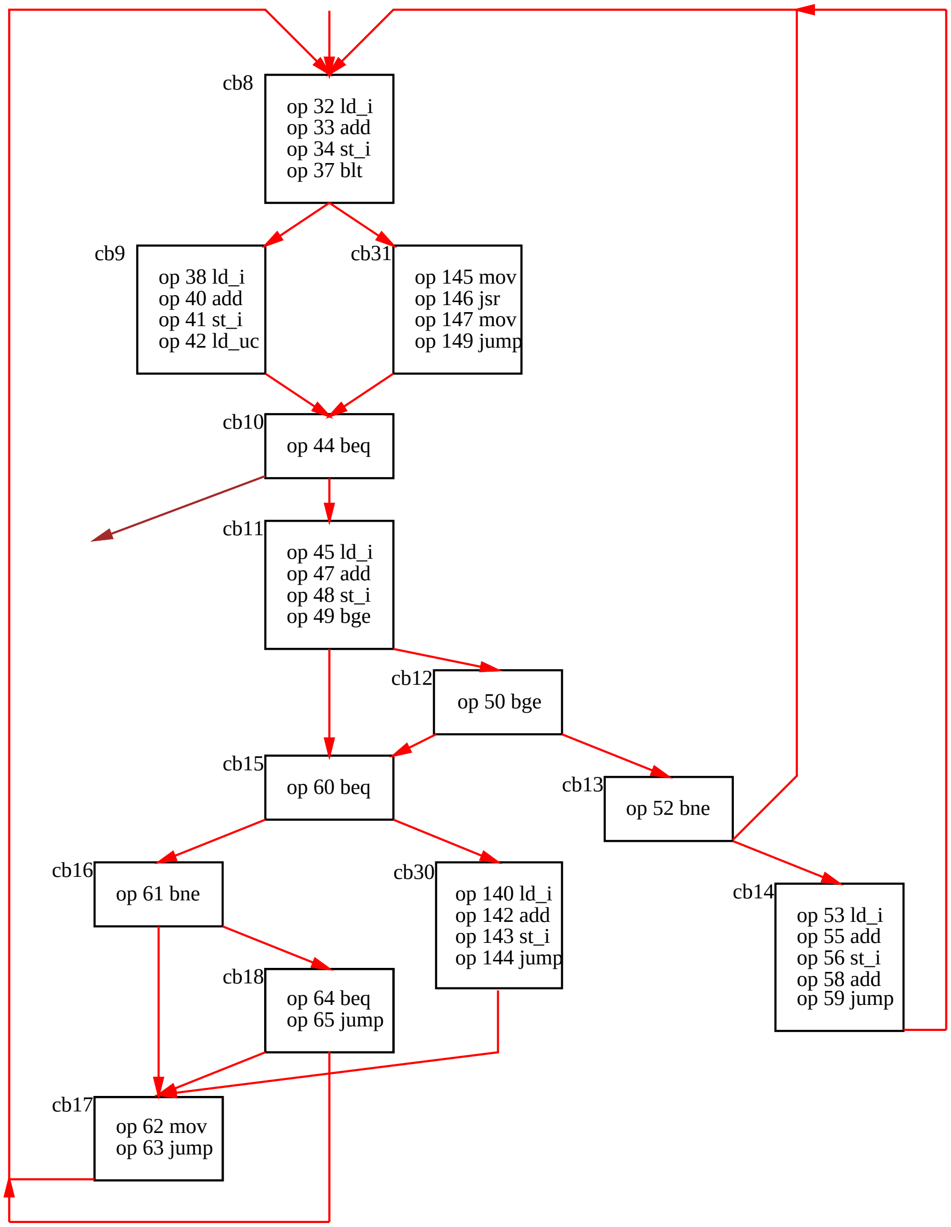
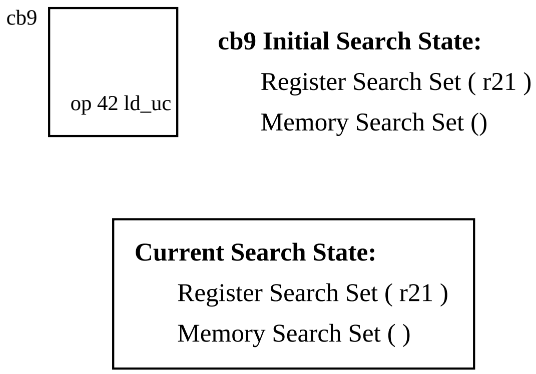


Figure 5.12 Initial wc loop prior to use-def graph generation.

```
op 32 ld_i r98,(r3 + 0)
op 33 add r18,r98,-1
op 34 st_i (r3 + 0),r18
op 37 blt r98,1,cb31
op 38 ld_i r21,(r3 + 4)
op 40 add r20,r21,1
op 41 st_i (r3 + 4),r20
op 42 ld_uc r4,(r21 + 0)
op 44 beq r4,-1,cb25
op 45 ld_i r24,(_charct + 0)
op 47 add r23,r24,1
op 48 st_i (_charct + 0),r23
op 49 bge 32,r4,cb15
op 50 bge r4,127,cb15
op 52 bne 0,r2,cb8
op 53 ld_i r27,(_wordct + 0)
op 55 add r26,r27,1
op 56 st_i (_wordct + 0),r26
op 58 add r2,r2,1
op 59 jump cb8
op 60 beq r4,10,cb30
op 61 bne r4,32,cb18
op 62 mov r2,0
op 63 jump cb8
op 140 ld_i r62,(_linect + 0)
op 142 add r61,r62,1
op 143 st_i (_linect + 0),r61
op 144 jump cb17
op 64 beq r4,9,cb17
op 65 jump cb8
op 145 mov param_reg,r3
op 146 jsr __filbuf
op 147 mov r4,ret_reg
op 149 jump cb10
```

Figure 5.13 Instructions for wc loop.



cb9 Initial Search State:
Register Search Set (r21)
Memory Search Set ()

Current Search State:
Register Search Set (r21)
Memory Search Set ( )

Figure 5.14 Initial node of use-def graph containing instruction 42.

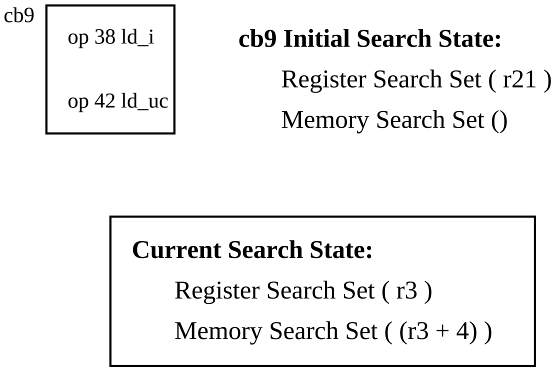

cb9
op 38 ld_i
op 42 ld_uc
cb9 Initial Search State:
Register Search Set (r21)
Memory Search Set ()
Current Search State:
Register Search Set (r3)
Memory Search Set ((r3 + 4))

Figure 5.15 Final state of node after traversing all instructions in cb 9.

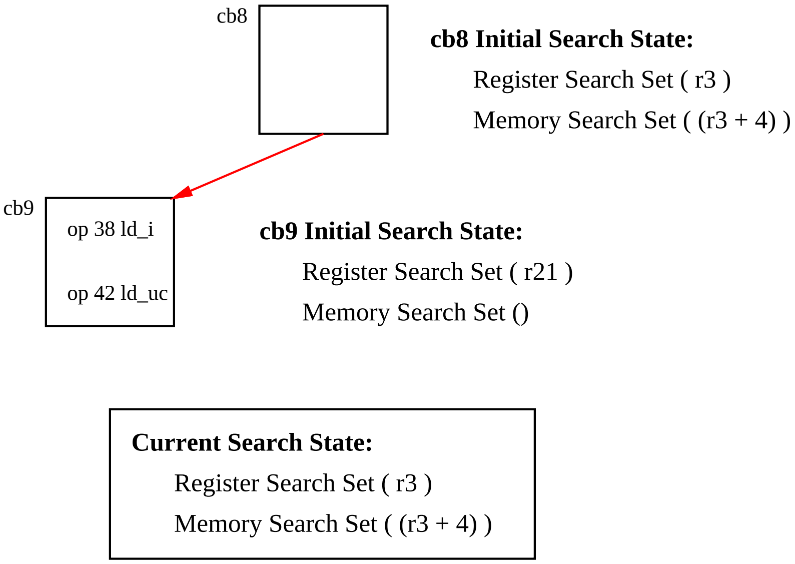

cb8
cb8 Initial Search State:
Register Search Set (r3)
Memory Search Set ((r3 + 4))
cb9
op 38 ld_i
op 42 ld_uc
cb9 Initial Search State:
Register Search Set (r21)
Memory Search Set ()
Current Search State:
Register Search Set (r3)
Memory Search Set ((r3 + 4))

Figure 5.16 Initial state of use-def graph prior to traversing cb 8.

4), is added to the memory search set since it ultimately determines what will be written to register 21 as a result of the load.

The only source arc to cb 9 is from cb 8. Figure 5.16 shows the state of the use-def graph prior to traversing into the cb. Since cb 8 does not exist in the use-def graph, a new node is created. Its initial register and memory search sets will be set to the contents of the current register and memory search sets. Next, the Recursive_Build_Use_Def_Graph will begin processing cb 8.

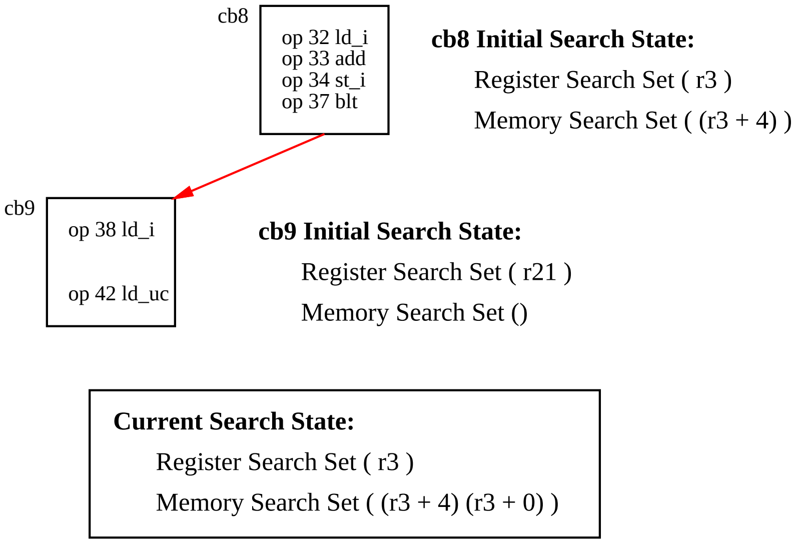

cb8
op 32 ld_i
op 33 add
op 34 st_i
op 37 blt
cb8 Initial Search State:
Register Search Set (r3)
Memory Search Set ((r3 + 4))
cb9
op 38 ld_i
op 42 ld_uc
cb9 Initial Search State:
Register Search Set (r21)
Memory Search Set ()
Current Search State:
Register Search Set (r3)
Memory Search Set ((r3 + 4) (r3 + 0))

Figure 5.17 Final state of use-def graph after traversing cb 8.

Figure 5.17 shows the state of the use-def graph after traversing cb 8. The branch, op 37, is added to the node list to keep track of all conditional paths. This results in the addition of register 98 to the current register search set. The store, op 32, is added to the list since there is an unknown pointer address in the memory search set [2]. Register 18 is added to the current register search set. Instructions 33 and 32 are added to the instruction list for the current node since their destination registers are in the current register search set. The resulting search state contains only register 3 and the memory addresses for (r3 + 0) and (r3 + 4).

Figure 5.18 shows that state of the graph several recursive calls later after traversing down one of the paths of loop control structure. Since the graph currently contains a node for cb 9, the first step is to determine if a new node is required. Since the current search state is not a subset of the initial search state when cb 9 was entered, we know that no traversal has passed along this path search for declarations of these operands. As such, a new node will be added

[2] The disambiguation model used is limited in the current implementation and does not differentiate between pointer addresses at this point. To prove that these two addresses do not conflict, the access patterns of the loads would need to be proven to be disjoint. In this case, the base address is invariant in the loop which is sufficient to show that these two instructions reference different memory addresses.

to the graph for cb 9. This graphs initial search state will be set to the current search state and traversal through cb 9 will begin again. If the current state had been a subset of the initial state of the node, then an arc would have been added to the current node representing cb 9 and traversal would have terminated along this path.

The traversal process will iterate until all paths have been followed for all definitions that are alive along these paths, thus reaching a steady state. Since there are only a finite number of instructions in the program, this will ultimately converge. Unfortunately, because of loops, this could result in the generation of one node in the graph for each cb in the function. Additionally, each cb could be represented once for each possible combination of memory references and register operands. Therefore, it is necessary to combine nodes that represent the same cb and merge the instruction lists into the original program order. An alternative approach to graph merging would be to update the state of the existing node and insert new instructions into the list as they are encountered [3]. Finally, the cycles in the graph will be identified and the graph will be levelized for the graph evaluation process. Figure 5.19 shows the final form of the graph after these steps have been performed. The edges marked with B are back-edges that form cycles in the graph. The horizontal lines separate the levels within the graph.

5.2.2 Evaluating Use-Def Graphs

The objective of evaluating the use-def graph is to capture sufficient information to determine all initial values that reach instruction operands in question as well as variances in these values that can be determined through one pass around the cyclic edges in the graph. This

[3] In the current implementation, the graph is not merged which results in a significantly longer evaluation process.

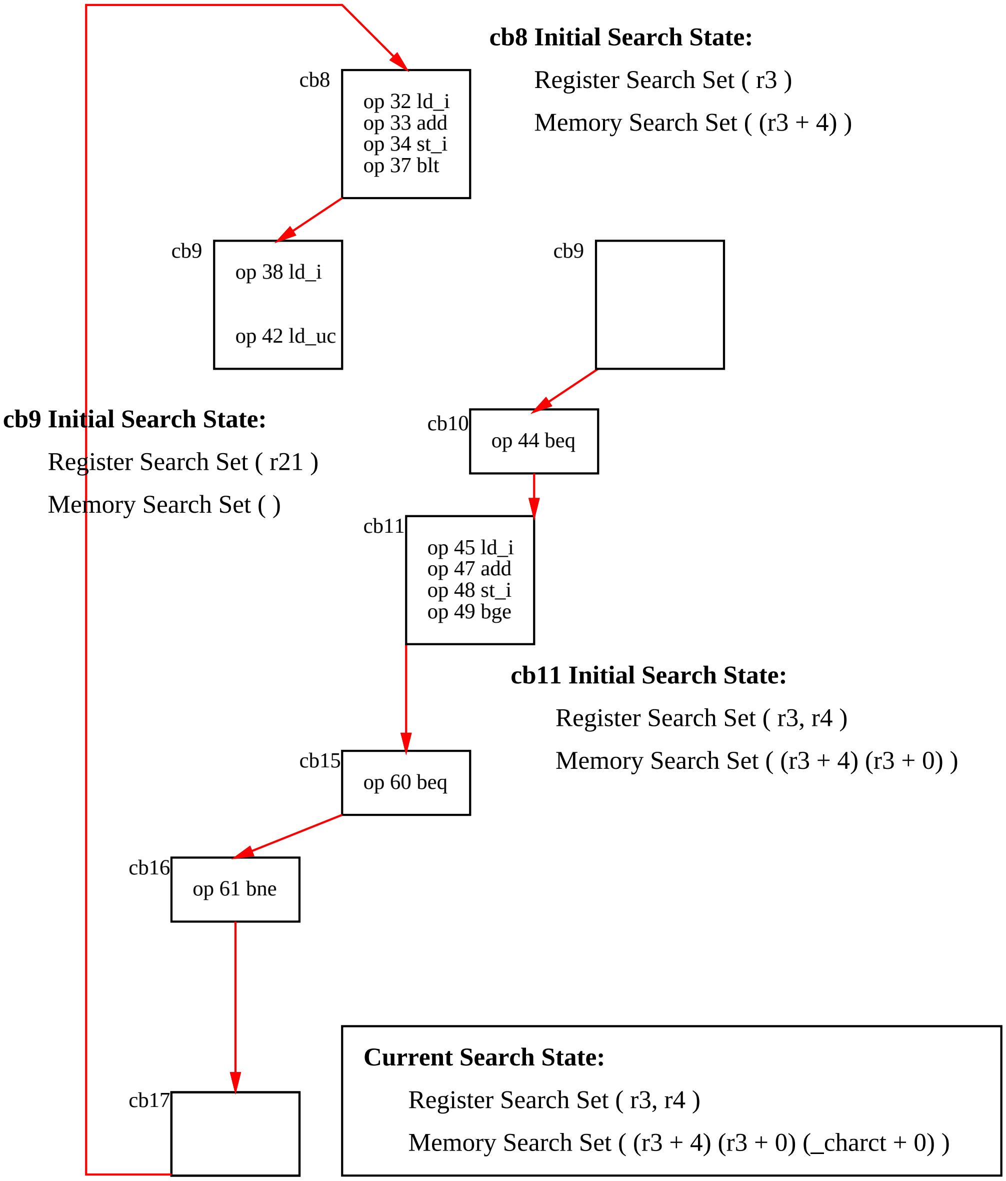

cb8 Initial Search State:
Register Search Set (r3)
Memory Search Set ((r3 + 4))
cb8
op 32 ld_i
op 33 add
op 34 st_i
op 37 blt
cb9
op 38 ld_i
op 42 ld_uc
cb9
cb9 Initial Search State:
Register Search Set (r21)
Memory Search Set ( )
cb10
op 44 beq
cb11
op 45 ld_i
op 47 add
op 48 st_i
op 49 bge
cb11 Initial Search State:
Register Search Set (r3, r4)
Memory Search Set ((r3 + 4) (r3 + 0))
cb15
op 60 beq
cb16
op 61 bne
cb17
Current Search State:
Register Search Set (r3, r4)
Memory Search Set ((r3 + 4) (r3 + 0) (_charct + 0))

Figure 5.18 State of use-def graph several recursive calls later.

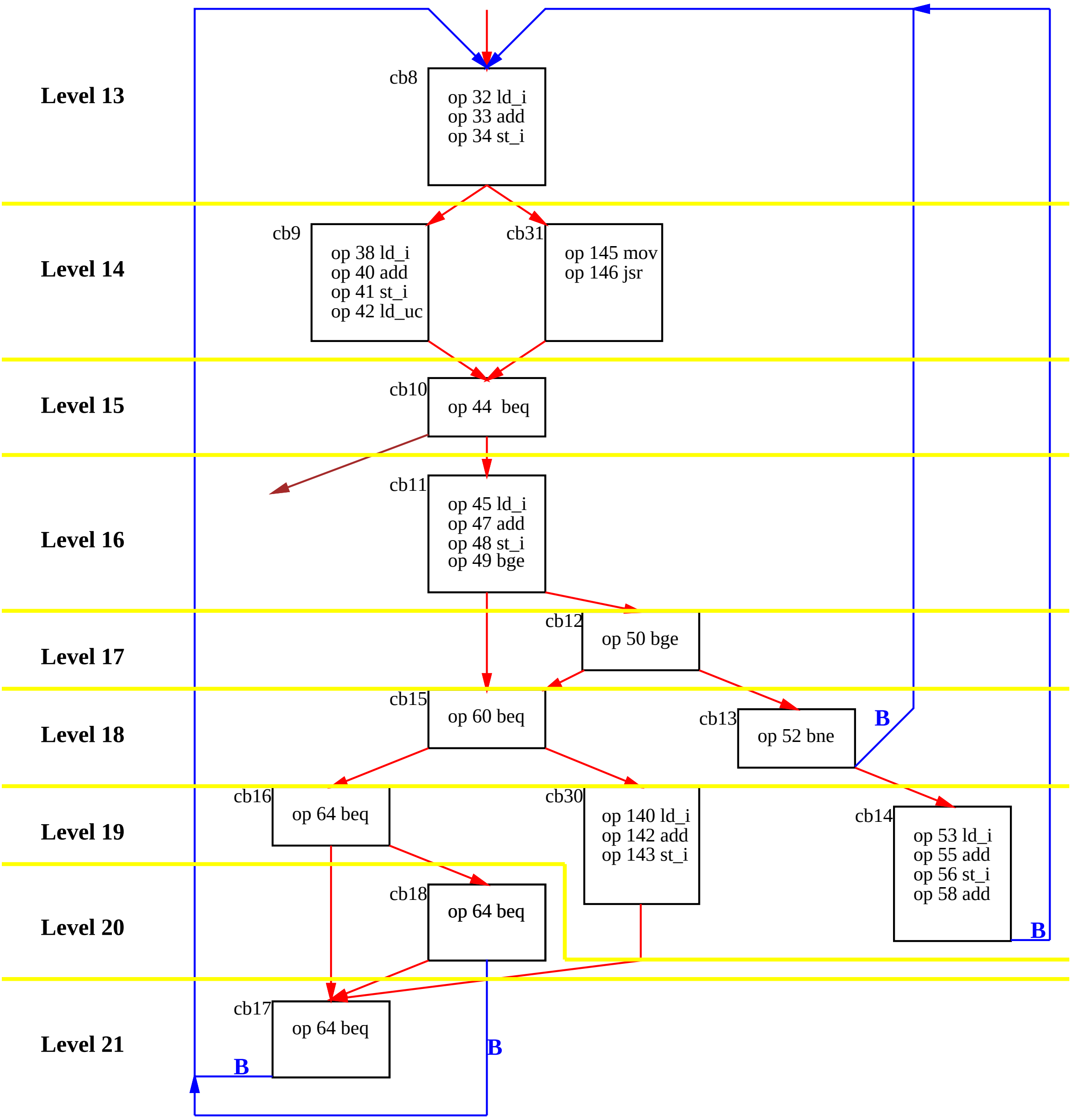


Figure 5.19 Final use-def graph after merging nodes, identifying cycles and leveling graph.

information will be used by later phases to determine loop constraints, as well as to determine the safety of potentially excepting instructions.

Before discussing the algorithms used to evaluate the graph, the concept of state must be introduced. The evaluation process contains two classes of state, program level and function level. Program level state contains the global memory state that was generated as a result of static initialization and one pass through all global variables in the program. Additionally, the parameter register and return register states of functions will be generated if inter-procedural analysis is being performed. These two categories of program state define the only means that information can be passed between functions in Lcode.

In addition to the program state, a function-level state is grown from an initial undefined state each time a use-def graph is evaluated. The function state includes the register file and memory. The memory state is composed of global memory, local memory and pointer memory. Global memory will contain the subsets of the program global memory state that are referenced during the graph evaluation. The local memory state contains all stack-based references. The pointer memory contains all pointer references that could not be resolved to local or global memory, thus they represent ambiguous references.

As Figure 5.20 shows, there are two loops. The first loop makes a pass down the use-def graph to produce values that can be generated along the acyclic edges in the graph. This provides all initial values that reach each instruction. To accomplish this, the initial state for a node is generated using the function Update_State. Figure 5.21 shows a simple example of the principal required to update the state. The state of a node will be the combined state of the states generated by all nodes that reach it. Thus, the state for node 5 is the combined state from nodes 2 and 3. Update_State returns a flag indicating if their it was determined if the node should be traversed. Since the current use-def graph does not merge nodes, it is possible for only one path to reach a given node and that path have the condition that would prevent traversal. This condition will eliminate the need to call Evaluate_Node. Next, the function Evaluate_Node will evaluate all instructions in a node, generate a final state for the node and return any values that reached the instruction that started the search process. The second loop performs the same steps as the first loop but includes state information that was generated from the first step that can be propagated around cyclic edges in the graph. This provides the variance information for the loop.

Figure 5.22 shows the algorithm for Update_State. The algorithm looks through all source arcs for those of the desired type, acyclic or cyclic, to produce the correct state. If the state, produced from a particular source, contains only the value that would have prevented traversing that edge, then the branch is noted and state is not propagated across that edge [4]. If this is not the case, then the current state of the node is merged with the state from the incoming node.

Figure 5.24 contains the algorithm for Evaluate_Node. This algorithm evaluates all instruction in a node list starting at the top of the list. Instructions use the current register and memory state to produce a revised register and memory state. In order to maintain useful information around cyclic edges in the graph, the old values in the register file and memory subsystem will be maintained through a block. However, register and memory locations that are multiply defined within a single pass through the block will contain one new definition which is the final definition. In all cases, only non-duplicate values are maintained through-out the evaluation process. This correctly handles the situation where multiple paths can produce

[4] This permits identification of conditional branches that prevent exceptions conditions from occurring. It also eliminates the need to evaluate a node that will produce information that is not useful for the final solution set.

Algorithm Evaluate_Use_Def_Graph: Given a levelized use-def graph, evaluate the nodes in the graph to determine the values that reach the the terminal node in the graph. Each node is evaluated only once in each loop. Due to inter-procedural analysis, the calling routine may provide initial values for parameter registers.

```
initialize parameter register state

values = NULL

/*
 * Make first pass down graph evaluating operands using values from
 * acyclic arcs.
 */
for each level in graph
{
    for each node at current level
    {
        traverse = Update_State ( node, ACYCLIC edges )

        if (traverse)
            values = Evaluate_Node ( node, values )
    }
}

/*
 * Make second pass down graph evaluating operands using values from
 * cyclic and acyclic arcs.  This ensures that values from back-edges
 * are propagated through loop.
 */
for each level in graph
{
    for each node at current level
    {
        traverse = Update_State ( node, ACYCLIC and CYCLIC edges )

        if (traverse)
            values = Evaluate_Node ( node, values )
    }
}
```

Figure 5.20 The Evaluate_Use_Def_Graph algorithm is generate all values that reach the source operands of the instruction that terminates the graph.

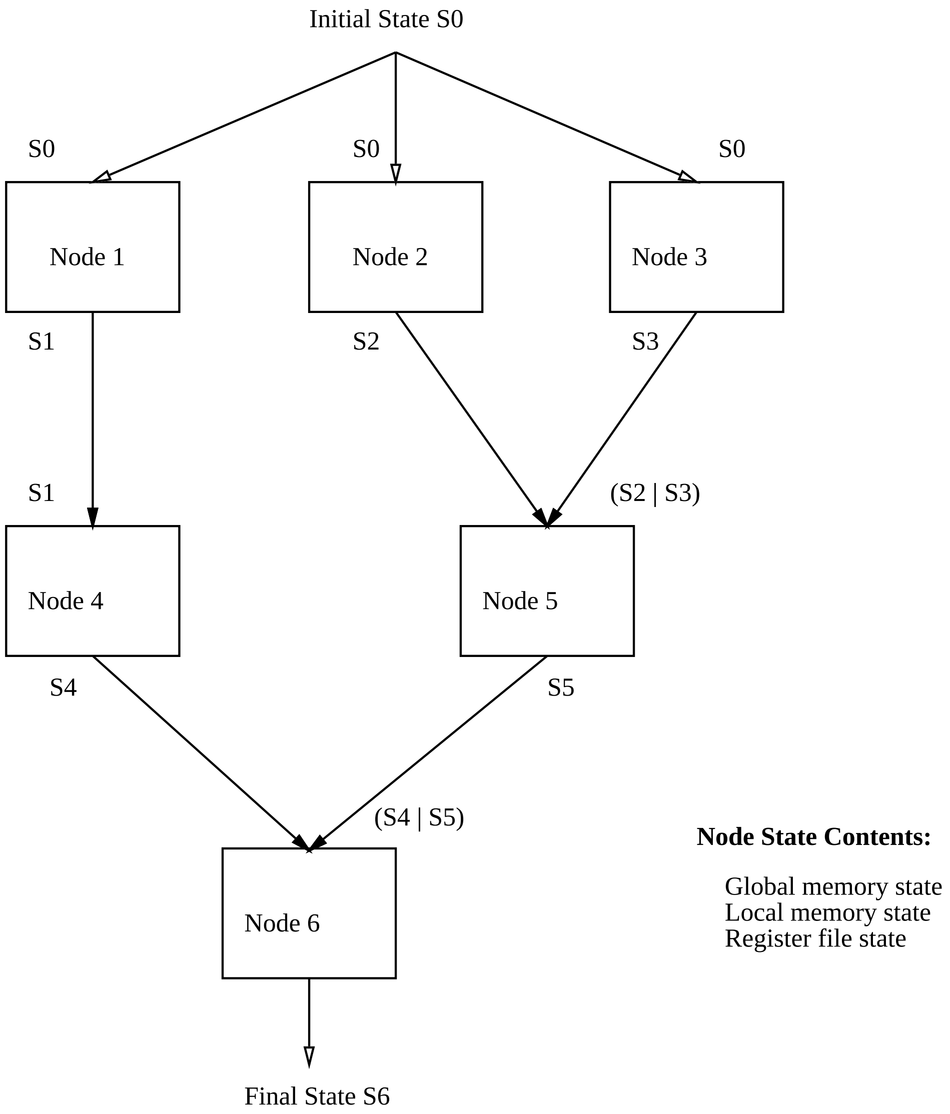


Figure 5.21 Example of how function state is migrated through a use-def graph.

Algorithm Update_State: Given a node in a use-def graph and the types of edges that are relevant to the graph search, generate the new state for the current node.

```
edge_traversed = 0
for each source_arc to current node
{
    if (source_arc->type is not of the desired type)
        continue

    source_node = source_arc->node

    if (source_arc would not be traversed as a result of its state)
    {
        remember the id of the branch that prevented traversal
    }
    else
    {
        node->register_state is updated by register state of source_node

        node->local_mem_state is updated by local memory state of source_node

        node->global_mem_state is updated by global memory state of source_node

        node->pointer_mem_state is updated by pointer memory state of
            source_node

        edge_traversed = 1
    }
}
return edge_traversed
```

Figure 5.22 The Update_State algorithm is used to compute the register and memory state that reaches the node.

multiple values but a single block can only produce one result per register for each path that reaches it.

The first requirement of the evaluation loop is to determine if the current instruction in the block is the terminal instruction. If it is, then the current values for the operands are recorded. The state of the operands are recorded prior to evaluating the terminal instruction in the event that it modifies one of its own source operands such as the case of a post-increment load.

Next, the different possible instruction types are evaluated using the current state. Based upon the evaluation requirements, instructions are broken down into six groups: move, store, load, arithmetic, subroutine calls and conditional branches. In the case of the move, values from one register file location are moved to another register file location. Source registers that have no defined values will result in adding an undefined value to the destination register. This ensures that conservative estimates are used through-out. Querying the register file can initiate the inter-procedural process if the source operand is an incoming parameter and inter-procedural analysis is being performed. As Figure 5.23 shows, a query for the incoming parameter in Func_D will initiate evaluation of the calling functions Func_A, Func_B and Func_C starting at the jsr call. If the inter-procedural processes results in any undefined results, then undefined results will be written to the register file. A more detailed discussion of this will be provided later.

Load instructions will require computation of all possible source addresses for the memory access. Memory accesses are divided into three groups: local, global and pointer. In the case of local accesses, the local memory state is queried. This state will only contain values that are defined within the scope of the function. Global access will initially go to the function-level global memory. If the contents are undefined, then the program-level global memory will

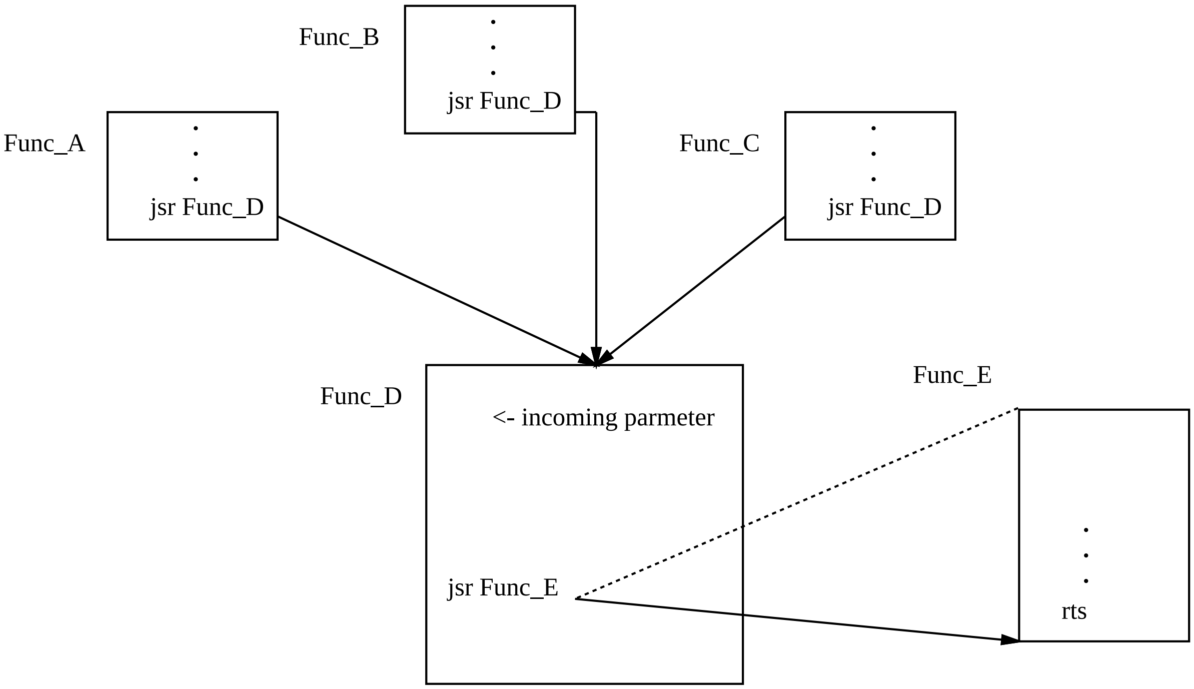


Figure 5.23 Examples of how inter-procedural analysis is performed.

be accessed. If this is undefined, then inter-procedural analysis will be performed. Ultimately, useful values or undefined values will be written to the function's global memory thus eliminating the need to perform further inter-procedural analysis for this global memory location. In some cases, it is not possible to determine that a load is to local or global memory. Therefore, a pointer memory state is maintained as well.

Store instructions operate the same as the load for the memory addresses computations and the same as move instructions for register file accesses. The only difference is that memory updates only go to the function-level memory state. They are not written through to global memory. Evaluating arithmetic instructions require applying the arithmetic operation on the values available from the source operands. If either of the values for source operands are undefined, then an undefined result is written to the register file. If either of the values would cause an error such as a divide-by-zero, then an undefined result is written to the register file.

Conditional branches provide a useful method of adding or eliminating values from the current register file contents. If the branch will be taken, then the condition that caused the branch to be taken must exist. For example, if the branch is taken if the register contents are equal to zero, then any values (including undefined values) that are present in the register can be eliminated that are non-zero. If the zero-value was not present, then it can be added. Since the current implementation of the use-def graph contains all enumerated paths, this required only updating the current state. However, by merging the graph to reduce its size, as discussed in Section 5.2.1, support for this will be more complicated. Since there are two states generated by the branch, one for the taken case and one for the not-taken case, it is necessary to produce two output memory states.

The algorithm shown in Figure 5.25 is provided as a more detailed example of the evaluation performed. This algorithm also shows the fundamental approach used to perform inter-procedural analysis when a subroutine call is encountered. In the event that inter-procedural analysis is not being performed, then any values for the subroutine return register that were defined within the current block but from other subroutine calls will be deleted and an undefined value is added to the register file.

If inter-procedural analysis is being performed, then the parameter registers are read from the current register file and a parameter state is created containing the values for only those parameter registers. The dotted line in Figure 5.23 represents the state being passed from Func_D to the entry point of Func_E. Next, any values for the subroutine return register that were defined within the current block but from other subroutine calls will be deleted to prevent propagation of old data. Finally, each possible subroutine that can be called from the current instruction is evaluated started at the *rts* in the target function. In an effort to reduce the

Algorithm Evaluate_Node: Given a node in a use-def graph as well as the register file and memory state, generate the state upon evaluation of all instructions in the node. Upon encountering the terminal instruction, the values of its source operands are recorded.

```
for each instruction in the current node
{
    if (instruction is the terminal instruction)
    {
        record source operand values for terminal instruction
    }

    if (move instruction)
    {
        Evaluate_Move ( node, instruction, state )
    }
    else if (load instruction)
    {
        Evaluate_Load ( node, instruction, state )
    }
    else if (store instruction)
    {
        Evaluate_Store ( node, instruction, state )
    }
    else if (arithmetic instruction)
    {
        Evaluate_Arithmetic ( node, instruction, state )
    }
    else if (conditional branch instruction)
    {
        Evaluate_Conditional_Branch ( node, instruction, state )
    }
    else if (subroutine call instruction)
    {
        Evaluate_Subroutine_Call ( node, instruction, state )
    }
}

return recorded values
```

Figure 5.24 The Evaluate_State algorithm is used to record the register and memory state changes produced by a node in the use-def graph.

execution time of the evaluation process, subroutine calls from a particular instruction to a specific function are only performed one time for each set of parameter values. Any subsequent calls with the same parameter values will be provided the previous return values from that invocation.

In order to perform the inter-procedural analysis, the return instruction from the subroutine call is identified as the starting point in the called function. The Build_Use_Def_Graph function is called with the starting instruction and the return parameter register as the operand which needs values. Then Evaluate_Use_Def_Graph is called to evaluate the graph and produce the return values from the function. Finally the current register state will be updated with these values. One important observation is that calling Evaluate_Use_Def_Graph results in an a recursive cycle in the evaluation process which provides the inter-procedural benefits. Without constraints, this iterative evaluation could result in traversals through the entire program. To reduce the overhead of analysis, a maximum of 5 levels of evaluation are permitted inter-procedurally from any given node in the call-graph.

Ultimately, the success of the inter-procedural analysis is dependent upon the availability of the source code for the libraries. Source code was unavailable for a number of critical library routines shown in Table 5.1. Therefore, the functionality of these routines were implemented to support the evaluation processes. For example, a call to malloc will create dynamically allocated buffer of the size specified. A call to fopen will return a pointer to a file structure that has its buffer count initialized to zero and the pointer to the character buffer initialized to zero. A call to __filbuf will take the file structure pointer and create a dynamically allocated buffer and initialized the buffer count to the size of the buffer.

Algorithm Evaluate_Subroutine_Call: Given a subroutine call, a node in a graph and the current state, evaluate the subroutine call to evaluate its effects to the current state.

```
return_reg = Get_return_registers ( instruction )

if (analysis_level != INTER_PROCEDURAL)
{
    delete register values defined through this pass of the block for
        return_reg

    Add_undefined_value_to_register ( current_state, return_reg )

    return
}

parameter_registers = Get_parameter_registers ( instruction )

parameter_state = Get_all_register_values ( current_state, parameter_registers )

delete register values defined through this pass of the block for return_reg

for each callsite of the current subroutine call instruction
{
    if ( callsite has not been called with the current parameter state )
    {
        return_instruction = Get_return_instruction ( callsite )

        graph = Build_Use_Def_Graph ( callsite, return_instruction,
            parameter_registers )

        values = Evaluate_Use_Def_Graph ( graph, parameter_state )

        update current_state with values returned from graph evaluation
    }
    else
    {
        update current_state with values returned from the previous
           evaluation containing the same parameter state
    }
}
```

Figure 5.25 The Evaluate_Subroutine_Call algorithm is used to record the register and memory state changes that occur as a result of traversing into a subroutine call.

fopen	__filbuf	malloc	read	fstat	calloc
realloc	stat	fdopen	freopen	popen	tmpfile

Table 5.1 Library routines that were implemented to support the analysis process.

5.2.3 Generating Call-Graphs for C Programs

An accurate call-graph is critical for inter-procedural analysis. Call-graphs that do not contain all arcs to functions that are reached by each subroutine call can result in missing information that could ultimately result in speculating potentially excepting instructions that are not actually safe. On the other-hand, if the graph adds arcs to functions that are not reachable in the original program, information may be provided that prevents identification of safe potentially excepting instructions that are actually safe. Thus, the graph needs to contain accurate arcs and only add extra arcs for subroutine calls that can not be resolved.

Programs written in C may contain subroutine calls of two forms: explicit and implicit. The target of an explicit subroutine calls is visible from the instruction. However, the target of an implicit call may be provided from a variable in the program. This variable may be initialized within the scope of the function or passed as a parameter to the function containing the implicit subroutine call. Both of these cases are supported in FORTRAN [47]. Unlike in FORTRAN, the variable may be a global variable [48]. This variable may even be defined as an array of function pointers. Thus, to generate C call-graphs in the presence of implicit subroutine calls, analysis of local function state, the parameter state and the global memory state are required. Since this problem is dependent on the values that reach the implicit subroutine calls, it can be solved using the infra-structure presented in sections 5.2.1 and 5.2.2.

Figure 5.26 presents the algorithm used to build C program call-graphs. The first loop in the algorithm constructs nodes and arcs for explicit subroutine calls. It also adds arcs from implicit subroutine calls to an unknown node for later evaluation. The second loop iterates through the program to resolve implicit subroutine calls. The function Build_Use_Def_Graph is passed the subroutine call instruction and the unknown operand for the graph building process. Then

Evaluate_Use_Def_Graph evaluates the use-def graph to determine the values that reach the function. Since the evaluation processes can perform inter-procedural analysis, the call-graph building processes can actually requires an accurate call-graph to completely resolve implicit subroutine calls. Thus, the call-graph building process should iterate until no paths are created that reach implicit subroutine calls. The current implementation will emit an error if such a condition occurs.

5.2.4 Identifying Side-Effect Free Subroutine Calls

In order to assure accuracy during the use-def graph generation phase, it is necessary to add subroutine calls to the traversal list if the subroutine can change update memory and there are any memory addresses in the memory search set. To reduce necessary subroutine traversal, it is necessary to identify subroutines that do not write to memory and are thus side-effect free.

Figure 5.27 contains the algorithm used to identify side-effect free subroutine calls. This algorithm begins by identifying C library subroutine calls that are side-effect free using the list is given in Table 5.2 [5]. The objective of the second loop is to propagate state from leaf side-effect free functions upward through the graph. Thus, every time a function is identified as side-effect free, it will force another iteration through the call-graph. In order for a function to be side-effect free, it must only contain subroutine calls to side-effect free functions and must not write to non-local memory.

5.2.5 Determining Safety of Instructions

Figure 5.28 shows the main algorithm for safety analysis used in this research. The first step is building the program call-graph as discussed in Section 5.2.3. The second step is to identify

[5] This step was required since source code for the library routines was not available.

Algorithm Build_Call_Graph: Given a program, generate a program call-graph.

```
for each function in the program
{
    current_node = Find_node ( function )
    for each instruction in the function
        if (subroutine call)
        {
            target_node = Find_node ( instruction )
            if (explicit target_node)
                add arc from current_node to target_node
            else
                add arc from current_node to unknown_node
        }
}

for each function in the program
{
    current_node = Find_node ( function )
    for each instruction in the program
    {
        if (subroutine call with unknown_callsite)
        {
            graph = Build_Use_Def_Graph ( current_node, instruction,
                instruction_operands )
            values = Evaluate_Use_Def_Graph ( graph, NULL )
            if (values)
            {
                for each returned_value in values
                {
                    target_node = Find_node ( returned_value )
                    add arc from current_node to target_node
                }
                if (no unknown values returned)
                {
                    remove arc to unknown_node
                }
            }
        }
    }
}
```

Figure 5.26 The Build_Call_Graph algorithm is to generate a program call-graph.

Algorithm Identify_Side_Effect_Free: Given a program call-graph, identify functions that do not write to global memory, thus permitting greater optimization and scheduling freedom.

```
for each function in the call-graph
{
    for each instruction in function
    {
        Mark any subroutine calls in the standard list as side-effect free
    }
}

change = 1
while (change)
{
    change = 0
    for each function in the call-graph
    {
        for each instruction in the function
        {
            if ((store instruction) &&
                (not local memory reference))
            {
                mark function as not side-effect free
                skip to next function in call-graph
            }
            if ((subroutine call instruction) &&
                (function being called is not side-effect free))
            {
                mark function as not side-effect free
                skip to next function in call-graph
            }
        }
        mark function as side-effect free
        change = 1
    }
}
```

Figure 5.27 The Identify_Side_Effect_Free algorithm is used to mark subroutine calls that are to functions that do not modify global memory or memory that is passed in as a parameter.

strlen	strcmp	strncmp	feof	fclose	perror
isalnum	iscntrl	isdigit	isgraph	islower	isprint
ispunct	isspace	isupper	isxdigit	tolower	touper
atoi	atol	strchr	strrchr	strstr	strcasecmp
strncasecmp	index	rindex	strpbrk	strspn	strcspn
clearerr	ferror	fgetpos	ftell	rand	srand
memchr	memcmp	setjmp	exit	exp	exp2
exp10	log	log2	log10	pow	sqrt
sin	cos	tan	asin	acos	atan
atan2	sinh	cosh	tanh	asinh	acosh
atanh	abs	labs	fabs	cabs	floor
ceil					

Table 5.2 Standard list of side-effect free subroutine calls found in the C libraries.

side-effect free subroutine calls as discussed in Section 5.2.4. Next, the global memory state for the program is built. This entails evaluating every global store in the program to determine all values that reach the store. This provides a conservative estimate of the contents of global memory to eliminate the need for inter-procedural analysis every time a global load is encountered. The initialization process utilizes the same algorithms as discussed in Sections 5.2.1 and 5.2.2.

After the setup overhead is complete, safety analysis is performed. The analysis is broken into four steps. The first step identifies all safe instructions that can be identified by operand inspection. From this point on, these instructions will be classified as trivially safe. The second step is to identify instructions that are safe due to an equivalent arithmetic instruction as proposed by Kennedy. The third step entails identifying non-loop instructions and loop invariant instructions that are safe. Finally, loop variant safety analysis is performed. The ordering given is followed to identify instructions as safe using the simplest analysis processes to reduce the analysis overhead. Each of these steps will be discussed in greater detail with their supporting algorithms.

Identification of trivially safe instructions is the most straight forward process. Figure 5.29 shows the algorithm utilized to determine if these instructions are safe. The function traverses

Algorithm Identify_Safe_PEI: Given a program call-graph, identify all potentially excepting instructions in the program that are safe for speculation.

```
Build_Call_Graph for the program

Identify_Side_Effect_Free functions

conservatively build global memory state

Identify_Trivially_Safe instructions

Identify_Equivalence_Safe instructions

Identify_Non_Loop_and_Invariant_Safe instructions

Identify_Variant_Safe_Instructions
```

Figure 5.28 The Identify_Safe_PEI algorithm is the main calling routine used to identify instructions that are safe for speculation with a program.

through each function in the call-graph and through each instruction in each function. If any instruction falls into the class of potentially excepting instructions, then its operands are evaluated to determine the safety of the instruction. One observation that can be made is that any of the instructions that fall into the category of trivially safe are instructions that are automatically generated by the compiler. Thus, in the case of a load, verifying that a constant offset from a base address is legitimate is not required since the only way that it could be illegal is if the compiler has a bug. However, the presented algorithm discusses the safety analysis in order to provide a sense of security.

A load can be identified as safe if the base address is an explicit global variable or a stack address. As long as the offset is a constant value, it is easy to determine the validity of the load reference. In the case of the global reference, this is accomplished by querying the structure maintaining the size information for global variables to determine the size. Thus, if the offset is less than the size, the global load is safe. In the case of a stack load, the only requirement is

Algorithm Identify_Trivially_Safe: Given a program call-graph, identify all instructions that can be determine safe from their explicit operands.

```
for each function in the program call-graph
{
    for each instruction in the function
    {
        if (instruction is potentially excepting)
        {
            if (load instruction)
            {
                if ((explicit global memory reference) &&
                    (explicit constant offset) &&
                    (constant offset is valid))
                    mark instruction as safe

                else if ((explicit stack memory reference) &&
                         (explicit constant offset) &&
                         (constant offset is valid))
                    mark instruction as safe
            }
            else if ((integer divide) && (non-zero divisor))
                mark instruction as safe

            else if ((integer remainder) && (non-zero divisor))
                mark instruction as safe
        }
    }
}
```

Figure 5.29 The Identify_Trivially_Safe algorithm identifies instructions that can be identified as safe through explicit operand values.

that the offset of the load is less than the function stack frame. A divide or a remainder can be trivially identified as safe if the divisor is non-zero.

Remember that safety analysis is performed prior to the phases of code generation that can introduce machine specific instruction to transfer between integer and floating point register files as well as before register allocation. Thus, these classes of trivially safe instructions are not seen at this point of the analysis process. These instructions are marked safe when they are inserted by the code generator and register allocator.

Figure 5.30 is used to determine if an equivalent instruction to a potentially excepting instruction exists that will ensure that the potentially excepting instruction will not introduce an exception. While the concept of equivalence can be extended to any instruction it has only been implemented to handle the most common case load instructions. The algorithm iterates over every function in the call-graph and every instruction within the function. It only looks for an equivalent instruction within the same function. The requirements for safety are that both instructions have the same base address and that their offsets are both constants. Additionally, to ensure that the operands are the same, there must be no redefinitions between the first *equivalent* instruction and the load in question. Finally, for this implementation, it is required that the equivalent instruction dominates the load in question. This ensures that the base address is guaranteed to be the same as the load. A less strict assumption could search for equivalent instructions on all paths that reach the load. Since the base address for the equivalent instruction and the load are the same, a constant offset indicates that the loads are referencing the same structure. To maintain safety, the equivalent instruction will be identified as a sentry instruction for the equivalent instruction. As discussed in the overview to this chapter, a sentry instruction is an instruction that ensures that a subsequent potentially excepting instruction will not introduce an error.

The next step required to identify safe instructions utilizes the use-def analysis discussed in Sections 5.2.1 and 5.2.2. This class of safety analysis relies on the value contents that will reach an instruction to ensure its safety. Figure 5.31 presents the algorithm used to determine the safety of non-loop and loop invariant potentially excepting instructions. This algorithm is iterated over all functions in the call-graph and all potentially excepting invariant or non-loop instructions in the function. For each type of potentially excepting instruction, a use-def

Algorithm Identify_Equivalence_Safe: Given a program call-graph, identify all instructions that can be found safe as a result of equivalent instructions. For simplicity, this function assumes domination.

```
for each function in the program call-graph
{
    for each pei_instruction in the function
    {
        if (pei_instruction is potentially excepting)
        {
            for each instr in the function
            {
                if ((load instruction) &&
                    (instr base address is as pei_instruction) &&
                    (instr offset and pei_instruction offset are constants) &&
                    (instr dominates pei_instruction) &&
                    (no redefinitions of base address exists between
                     instr and pei_instruction))
                {
                    mark pei_instruction as safe

                    mark instr as sentry for pei_instruction
                }
            }
        }
    }
}
```

Figure 5.30 The Identify_Equivalence_Safe algorithm identifies instructions that can be identified as safe because of equivalent instructions that dominate the potentially excepting instruction.

graph is built to determine the content of unknown source operands. Then evaluation of the graph is performed to determine all possible values that reach the respective source operands. Additionally, for each type of potentially excepting instruction, it is critical that all returned values are known. In other words, no undefined values may reach the operands if the instruction is to be determined to be safe. To verify that a load is safe, the base and offset of the load must be computed. To ensure safety for each potential base and offset reference, the load with this combination of operands is passed to the Identify_Trivially_Safe algorithm. If each combination of base and offset is safe, then the load is marked as safe. In the case of integer divide and remainder, the only requirement is to verify that all values for the denominator are non-zero.

The final aspect of the safety analysis entails identifying loop variant potentially excepting instructions that are safe. The current implementation only addresses variant load instructions. As with the other analysis methods, the algorithm iterates over all functions in the call-graph. However, it only looks at variant instructions in the loops. The first step is to determine the loop bounds for the loop. The analysis process utilizes the same use-def generation and evaluation tools discussed so far. The standard impact libraries are used to identify loop branches and induction variables. This information is then used to determine loop initial values, final values and increments. When this information is identified, it is trivial to identify that the number of times that a loop will iterate. However, there are cases, such as the inner loop of wc, where the loop iteration bounds can not be determined. Additional loop analysis supports one special case that is used to determine the iteration space of the loop. If the exit branch is looking at the contents of a buffer or array for an exit value such as a such as a NULL or EOF marker as occurs when traversing through character buffers, it is possible to determine the iteration space of the loop.

Algorithm Identify_Non_Loop_and_Invariant_Safe: Given a program call-graph, identify all non-loop and loop invariant instructions that can be classified as safe from use-def graph traversal.

```
for each function in the program call-graph
{
    for each instruction in the function
    {
        if ((instruction is potentially excepting) &&
            ((instruction is not in a loop) ||
             (instruction is loop invariant)))
        {
            if (load instruction)
            {
                graph = Build_Use_Def_Graph ( current_node, instruction,
                    instruction_operands )

                values = Evaluate_Use_Def_Graph ( graph, NULL )

                if ((each returned value is known) &&
                    (each returned value ensures a safe load))
                    mark instruction as safe
            }
            else if ((integer divide) ||
                     (integer remainder))
            {
                graph = Build_Use_Def_Graph ( current_node, instruction,
                    divisor_operand )

                values = Evaluate_Use_Def_Graph ( graph, NULL )

                if ((each returned value is known) &&
                    (each returned value is non-zero))
                    mark instruction as safe
            }
        }
    }
}
```

Figure 5.31 The Identify_Non_Loop_and_Invariant_Safe algorithm identifies all non-loop and invariant instructions that can be determined safe through use-def traversal.

Algorithm Identify_Variant_Safe: Given a program call-graph, identify all variant load instructions that can be classified as safe from use-def graph traversal.

```
for each function in the program call-graph
{
    for each loop in the function
    {
        generate loop information use use-def traversal

        if ((loop bounds can not be determined) ||
            (loop bounds can be different values))
            continue

        for each instruction in a loop in the function
        {
            if ((instruction is potentially excepting) &&
                (instruction is variant))
            {
                if (load instruction)
                {
                    Identify_Variant_Safe_Loads ( current_node, instruction,
                        loop_information )
                }
            }
        }
    }
}
```

Figure 5.32 The Identify_Variant_Safe algorithm identifies all variant instructions that can be determined safe through use-def traversal.

As long as the loop worst-case trip count can be determined, the invariant safety analysis will be performed. The presence of multiple worst-case trip counts will be discussed in Chapter 7. A complication of the current analysis is the presence of loops whose iteration space is based upon linked list traversals or whose iteration space is determined by state transitions as present in parsers. These types of loops do not convey iteration space and as yet have not been studied to identify safety of variant loads.

Figure 5.36 shows the algorithm used to determine the safety of a load instructions whose address varies within the loop. First, a use-def graph is built for the potentially excepting load.

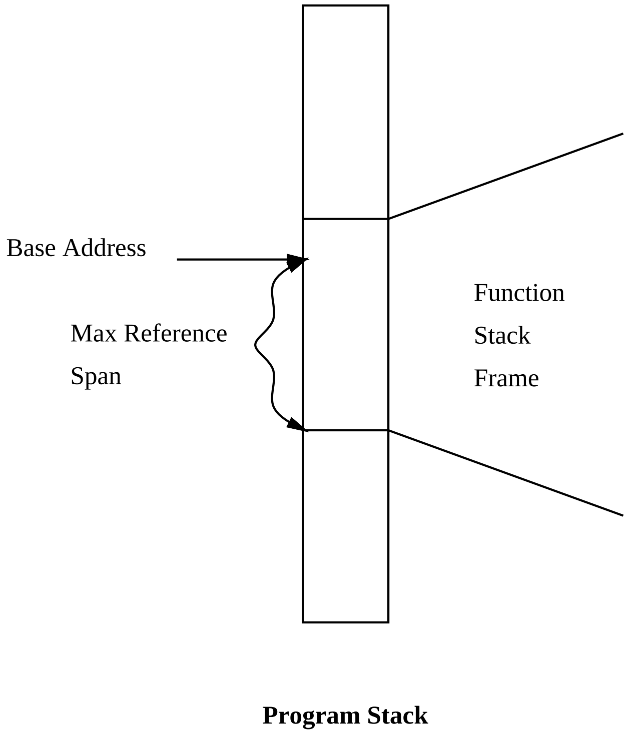


Figure 5.33 Memory reference size limitations for stack based loads.

Then the graph is evaluated to determine all values that reach the load. The value information is used to determine not only the base addresses of the loads but also the variance in the base address over one iteration of the loop. As long as all base addresses and offsets that reach the load are known, then safety analysis can proceed. For each base address and offset pair, a memory reference span for the loop is computed [6].

As Figure 5.33 shows, to determine that a stack based invariant load is safe, it is necessary to determine that the maximum span address is within the bounds of the stack frame. If this is possible, the load is marked safe. To maintain the safety of variant loads, optimizations constraints are required. In the case of stack loads, the load can vary through-out the function's stack frame without introducing an exception. The loop header is marked with the maximum

[6] In the current implementation, it is assumed that the load offset is incremented by the maximum offset value each iteration. More aggressive evaluation can identify the induction variable for the load to address cases where load offset is incremented on a different counter.

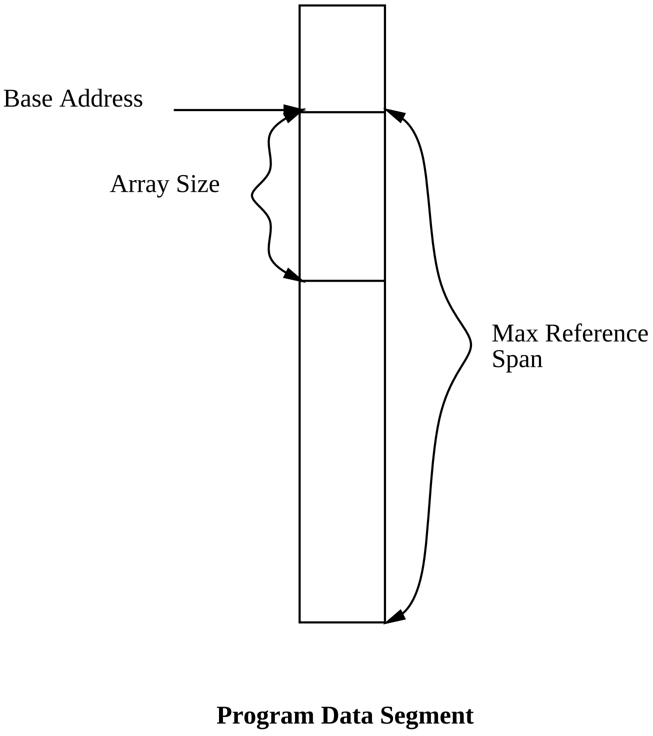


Figure 5.34 Memory reference size limitations for load from global memory.

number of references that are possible starting at the initial offset for the load to the end of the stack frame. This information will be used later to constrain optimizations such as loop-unrolling to ensure that the load remains safe.

As Figure 5.34 shows, a load from global memory requires the same analysis as a stack based load, however there are less restrictions for the reference pattern than stack based loads. The load reference may span through-out the entire global data segment without causing an exception. Thus, the loads reference pattern can not go beyond the end of the data segment [7]. One might questions that this assumption and the comparable assumption used for stack based loads fall into the category of general speculation. However, it can be countered that the only way that the load could have caused an exception in the first place is if its buffer were

[7] For this assumption to be valid, the data segment layout is assumed to be constant. Any type of data layout optimizations could negate the safety of this condition. Thus, the safety constraint would be to ensure that the memory reference pattern will stay within the bounds of the array being referenced.

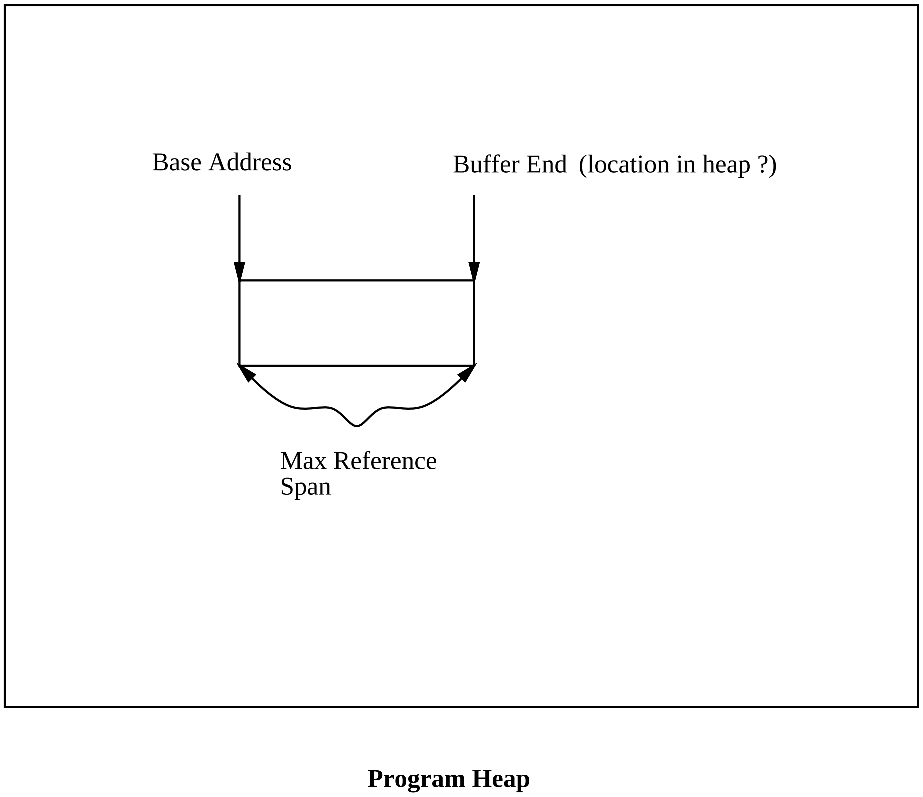


Figure 5.35 Memory reference size limitations for load from dynamically allocated memory.

located at the end of the data segment and the load reference pattern were to span outside the data segment causing an access violation. Thus, speculating the load will not hide an error or introduce an error as long as the reference pattern is still within the bounds of the data segment. As with the stack based load, the loop header is annotated with the maximum number of references that are possible from the base of the load to the end of the data segment in order to maintain safety.

Finally, as Figure 5.35 shows, a load from a dynamically allocated buffer utilizes the same analysis as the other two classes. However, while there are simplifying assumptions that can be used to reduce the access limitations for global or stack memory references, this is not the case for dynamically allocated memory. This is because the dynamically allocated buffers may be located anywhere in the heap and there is no way to determine statically where the buffer will be located within the heap. Thus, it is necessary that all memory references for dynamically allocated buffers fall within the bounds of the buffer. Additionally, the maximum reference pattern for the buffer load is limited to the size of the dynamically allocated buffer. This information is also marked in the loop header. Unlike the stack and global accesses, these references patterns limit optimizations such as unrolling to an factor of the buffer size. This problem is discussed in greater detail in Section 5.1.4.

Once all possible load references have been determined to be safe, the load will be marked as safe. In some cases, branch conditions exist that prevented invalid values from reaching the load. These branch conditions must be marked as sentry instructions to prevent the load from speculating above the branch. Finally, all of the maximum access patterns and the exact access patterns are evaluated to place constraints on the optimization phase to ensure that the load instructions will stay safe.

5.3 Experimental Evaluation

This section will quantify the benefits of the safety analysis used to identify potentially excepting instructions that are safe. The safety analysis techniques were implemented and then applied using intra-procedural and inter-procedural analysis to determine the merits of the analysis. These results were compared with no speculation, restricted speculation and general speculation to show the benefits and limitations. The results will be analyzed, deficiencies will be discussed and potential improvements will be proposed. Before presenting the results, however, the next section will show the the results from call-graph generation using the infrastructure presented in this chapter.

Algorithm Identify_Variant_Safe_Loads: Given a variant load instruction, and loop information, determine the conditions under which the load will be safe for speculation.

```
graph = Build_Use_Def_Graph ( current_node, instruction, instruction_operands )
values = Evaluate_Use_Def_Graph ( graph, NULL )

if (all base addresses and offsets are known)
{
    for each base address and offset pair
    {
        compute span of offsets that can be referenced in loop.
        if (base address is stack)
        {
            if (all of these offsets are within the function stack frame)
            {
                annotate loop header with MAXIMUM number of elements that can
                    to remain within the functions stack frame.
            }
            else
                instruction is unsafe, return
        }
        else if (base address is to global memory)
        {
            if (all of these offsets are within the bounds of the data segment)
            {
                annotate loop header with MAXIMUM number of elements that can
                    to remain within data segment.
            }
            else
                instruction is unsafe, return
        }
        else if (base address is to dynamically allocated memory)
        {
            if (all of these offsets will be within the bounds of the
                dynamically allocated buffer)
                annotate loop header with EXACT number of elements that can be
                    accessed
            else
                instruction is unsafe, return
        }
    }
    mark instruction as safe
    if (any branch conditions are preventing errors)
        mark branch condition as sentry for load
    use all EXACT and MAXIMUM access patterns to constrain optimization to
        maintain safety.
}
```

Figure 5.36 The Identify_Variant_Safe_Load algorithm identifies all variant load instructions that can be determined safe through use-def traversal.

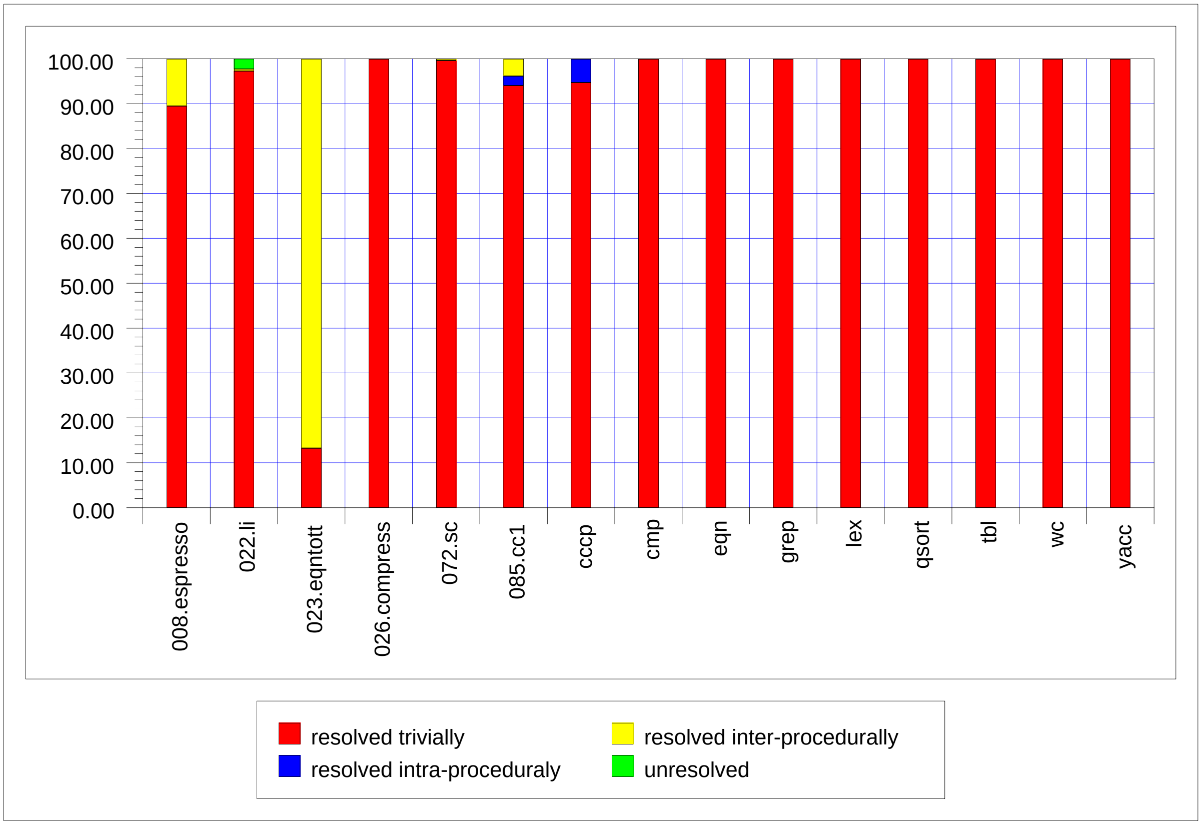


Figure 5.37 Experimental results from call-graph generation.

5.3.1 Experimental Results from Call-Graph Generation

The purpose of this set of experiments was to determine the difficulties in generating call-graphs. Figure 5.37 show the results of the experiments for the benchmarks shown in Table 3.1. The graph provides four results per benchmark. These results indicate the percentage of unresolved targets for subroutine calls based upon trivial operand inspection, intra-procedural analysis and inter-procedural analysis. The final piece of information indicates the percentage of subroutine calls that could not be resolved. These results are based upon the dynamic execution frequency of each subroutine call to provide the importance of the unresolved arcs.

The first observation is that with the exception of cccp, all of the non-SPEC benchmarks could be completely resolved through operand inspection. In the case of cccp, there were two unresolved subroutine calls in two separate functions which accounted for 5.14 percent of the dynamic subroutine calls in the benchmark. The targets of these subroutine calls were resolved intra-procedurally through a statically defined global array. To ensure accuracy, all function addresses from the array were added as arcs in the call-graph.

In contrast, only 026.compress in the SPEC integer benchmarks could be resolved completely by means of operand inspection. All of the remaining benchmarks required varying degrees of inter-procedural analysis to resolve the unknown subroutine calls. The benchmark 023.eqntott stands out from the rest of the benchmarks since 87 percent of the dynamic subroutine calls could not be resolved without inter-procedural analysis. The majority of this resulted from 11 of the 359 static subroutine call instructions in the program. The importance of this is that optimizations such as inlining would be limited in their potential benefits for this benchmark if detailed analysis were not performed to generate the call-graph. The front-end of the IMPACT compiler that performs inlining only uses operand inspection to build C call-graphs.

The only benchmark that the analysis was not able to fully resolve was 022.li. Approximately 3 percent of the dynamic subroutine calls could not be resolved. The dynamic percentages were contributed by 4 static subroutine calls out of the 1258 in the program. After analyzing the program, it was determined that the limiting factor was information provided by the input file. The subroutine called is dependent upon the expression read from the input file. Since the analysis process is value dependent, it was impossible to determine the subroutine for these callsites since there was no input file.

5.3.2 Experimental Results of Safe Speculation

The experimental results presented in Figure 5.38 present the relative speedup results of each of the speculation models simulated using an 8-issue superscalar processor over the same benchmark with no speculation using a single issue superscalar processor. The processor model used in this study is an in-order issue superscalar processor with register interlocking and 64 integer registers. The processor has uniform functional units, one branch delay slot and the instruction of the HP PA-RISC processor. The instruction latencies assumed are those of the HP PA-RISC 7100 presented in Table 3.2. The program executions were produced assuming perfect instruction and data caches. The performance results considering cache effects are presented in Chapter 6. In order to make the graph more readable, the speedup results for cmp using safe speculation with inter-procedural analysis and general speculation were clipped at 8. The actual speedup results for both benchmarks was 10.49 times faster than the single-issue no speculation performance results.

A summary of the results are presented in Table 5.3. Overall, restricted speculation with the addition of trivial safety analysis was 13 percent faster than no speculation at 8-issue. Safe speculation with intra-procedural analysis was able to attain a 20 percent speedup over restricted speculation. Safe speculation with inter-procedural analysis was able to attain a 55 percent speedup over restricted speculation and a 30 percent speedup over safe speculation with intra-procedural analysis. General speculation was 19 percent faster than safe speculation with inter-procedural analysis.

The results can be broken down further by separating the SPEC integer benchmarks from the UNIX benchmarks. In general the potential performance results for the SPEC benchmarks were lower than the UNIX benchmarks. Safe speculation with intra-procedural analysis was only 12

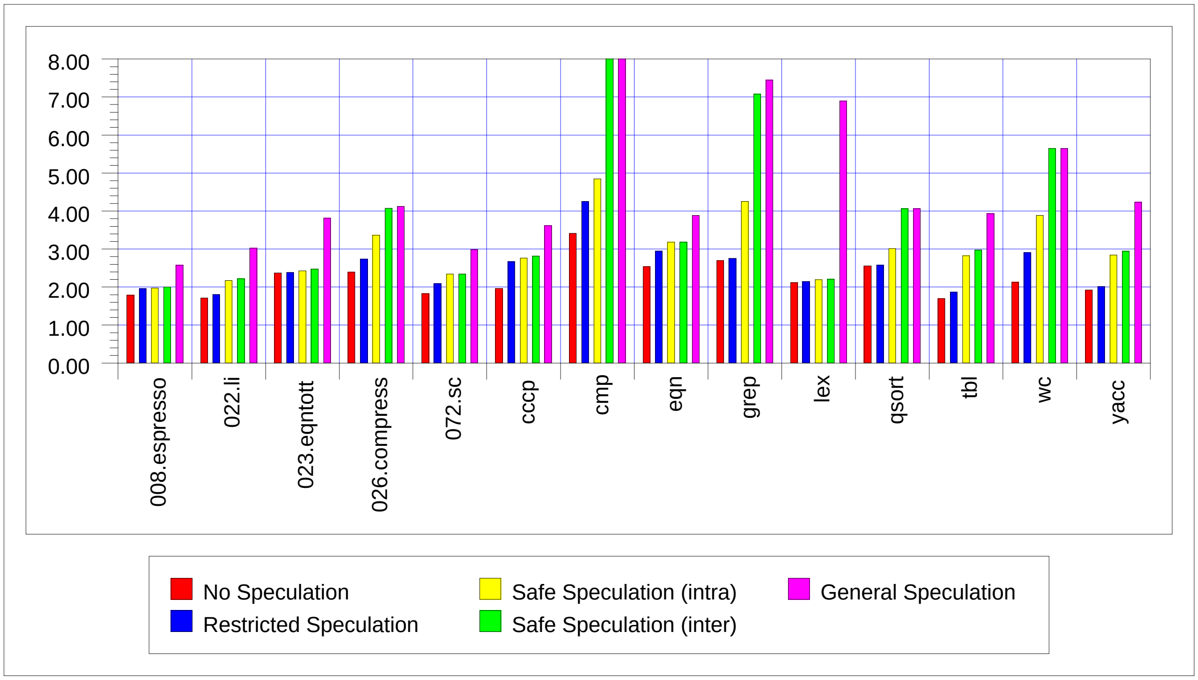


Figure 5.38 Experimental results comparing safe speculation with existing speculation models.

Speculation Model	Average of SPEC Int	Average of Non SPEC	Overall Average
None	2.02	2.34	2.23
Restricted	2.20	2.69	2.51
Safe (intra)	2.46	3.32	3.01
Safe (inter)	2.63	4.60	3.90
General	3.24	5.58	4.64

Table 5.3 Summary of speedup results for safe speculation and existing speculation models.

percent faster the restricted speculation and safe speculation with inter-procedural analysis was only 7 percent faster than with intra-procedural analysis. Most of the performance benefits were contributed by 026.compress. In contrast, there were more significant performance advantages with the UNIX benchmarks which showed a 23 percent speedup for safe speculation with intra-procedural analysis over restricted speculation while safe speculation with inter-procedural analysis demonstrated an average speedup of 39 percent over intra-procedural analysis.

The benchmarks wc, cmp, grep and 026.compress show common traits that demonstrate the advantages of the safety analysis proposed in this research. By exposing the operand values that could be identified statically, it was possible to identify the loop constraints for critical loops. The general performance advantages of inter-procedural analysis versus intra-procedural analysis demonstrate that there is a significant amount of information not visible within the scope of the function that provides critical information to determine the safety of potentially excepting instructions.

One common trait of wc, cmp and grep was that it was not possible to completely overlap the loop iterations of their critical loops after unrolling. This was because one or more loads within the loops were reached by an unsafe value that was produced outside the loop. In each case, there was a branch condition at the top of the loop to rectify the unsafe condition [8]. An optimization was performed to copy the branch condition and the correction code to the loop preheader to ensure that the buffer was always defined upon entering the loop. This permitted complete overlap of critical loops in wc and cmp that resulted in 76 percent speedup in cmp and 26 percent speedup of wc using inter-procedural analysis.

[8] One example was the file pointer buffer is NULL after a call to fopen. It will be initialized to a known state by a call to getc.

A second significant contribution in performance was permitted by taking advantage of relaxed memory reference constraints available when referencing global memory. By assuming that cache optimizations could be performed and thus constraining the buffer reference limitations to those of a dynamically allocated buffer, the performance 026.compress was reduced by 30 percent. Investigation showed that the stricter safety constraints limited the most critical loop in compress to an unroll factor that is a power of 2. The best unroll for the loop was a power of 3 which caused the performance loss. By removing this constraint, the loop was able to be unrolled by the desirable amount.

While it is desirable to present good performance results, it is also beneficial to understand the limitations of an approach. The SPEC benchmark 023.eqntott provides examples of the limitations of the current value-based solution to safety analysis. Figure 5.39 contains the source code for *cmppt*, the most frequently executed function in 023.eqntott.

The first observation from cmppt is that the loop iteration frequency is constrained by the variable *ninputs*. Unfortunately, the only value that reached this loop counter was undefined. As a result, it was not possible to perform any safety analysis for variant loads. Through further investigation, it was possible to determine that the variable could range anywhere from 1 to 127. Unfortunately, the range was dependent upon the contents of the input file which were unavailable. Given that the maximum trip count could vary over different invocations of the function, it would be unsafe to assume any fixed loop iteration count.

It is possible to overcome the lack of information about the iteration span of a loop by producing multi-version loops. Safety analysis could mark loop headers that require multi-version loops which would permit more aggressive safe scheduling. This solution not only addresses loops where it is not possible to determine the iteration span but also loops where

```
int cmppt (a, b)
PTERM *a[], *b[];
{
        register int i, aa, bb;
        for (i = 0; i < ninputs; i++) {
                aa = a[0]->ptand[i];
                bb = b[0]->ptand[i];
                if (aa == 2)
                        aa = 0;
                if (bb == 2)
                        bb = 0;
                if (aa != bb) {
                        if (aa < bb) {
                                return (-1);
                        }
                        else    {
                                return (1);
                        }
                }
        }
        return (0);
}
```

Figure 5.39 C source code for the function cmppt.

multiple worst-case trip counts occur. In many of these cases, the loop iteration frequencies were controlled by the size of arrays or buffers, but the varying sizes caused the safety analysis to be conservative and constrain unrolling to a common factor of all buffer sizes. If these buffer sizes had no common factors such as in buffers of size 256 and 125, then no unrolling is possible. In cases where these multiple sizes are constant, the information could be conveyed to the loop optimizer to generate loops optimized for each of the size constraints. Test code could then be inserted to ensure that the correct optimized loop is used.

The variables *a* and *b* are entries in the array *pts*. However, that is as much information that could be determined through the analysis, because any other information about the contents of these entries are dependent upon the input file and therefore each entry is undefined. In general, it is difficult to determine safety of variant loads if their definition state is based upon input files.

The final limitation observed resulted from loops that are difficult or impossible to determine worst-case trip count. The current loop analysis assumes that a single constant loop frequency can be determined to apply safety analysis to variant loads. However, parse loops such as found in the function *yyparse*, do not exhibit this type of behavior. More detailed analysis is required to determine that variant loads within these type loops are safe. Conceptually, it should be possible to determine the safety of many of these loads since state machines tend to be implemented in a table driven manner. The second complication to determining loop bounds are due to linked-list traversals. In these cases, it is not possible to determine constant loop bounds.

CHAPTER 6

A STUDY OF THE EFFECTS OF COMPILER-CONTROLLED SPECULATION ON INSTRUCTION AND DATA CACHES

6.1 Expected Cache Effects

This section will provide a qualitative analysis of the expected instruction and data cache effects resulting from compiler-controlled speculation. To accomplish this goal, scheduled code examples from two benchmarks are presented. These code examples were chosen because they show extreme cache effects due to speculation. The instruction opcodes and their descriptions for the examples are given in Table 6.1. The examples were scheduled with the no speculation and general speculation models using an eight-issue superscalar processor that has uniform functional units and instruction latencies of the HP-PA 7100 (see Table 3.2). The Icache and Dcache block sizes were 64 bytes each.

Icache Effects

Opcode	Description	Opcode	Description
add	32-bit integer add	ld_uc	load unsigned 8-bit value
beq	conditional branch on equal	ld_uc2	load unsigned 16-bit value
bgt	conditional branch on greather than	lsl	32-bit logical shift left
bne	conditional branch on not equal	mov	move 32-bit value
jump	unconditional branch	st_c	store 8-bit value
ld_c2	load signed 16-bit value	sub	32-bit integer subtract
ld_i	load signed 32-bit value	xor	32-bit exclusive or

Table 6.1 Instruction opcodes and descriptions.

Cycle	Instruction - (instruction id, opcode)				
1	45, ld_uc	48, mov	49, add	50, add	
2					
3	46, st_c	47, lsl	51, bgt	52, mov	
4	53, bgt	54, add	57, mov		
5	55, add	58, add			
6	56, ld_i				
7					
8	59, beq	60, mov			
9	61, add				
10	62, bne	66, ld_uc	63, lsl		
11	64, add				
12	68, lsl	67, st_c	65, mov	69, bgt	70, mov
13	71, bgt	72, add	75, mov		
14	73, add	76, add			
15	74, ld_i				
16					
17	77, beq	78, mov			
18	79, add				
19	80, bne	81, lsl	83, add	84, add	85, add
20	82, add	86, jump			

Figure 6.1 The most important loop in cccp scheduled using no speculation model.

Cycle	Instruction - (instruction id, opcode[* = speculative])							
1	45, ld_uc	54, add*	57, mov*	60, mov*	52, mov*	72, add*	48, mov	49, add
2	55, add*	58, add*	61, add*	50, add	73, add*	63, lsl*	75, mov*	70, mov*
3	47, lsl	46, st_c	66, ld_uc*	51, bgt	53, bgt	64, add*	76, add*	78, mov*
4	56, ld_i	79, add*	81, lsl*					
5	68, lsl*							
6	59, beq	62, bne	67, st_c	74, ld_i*	65, mov	69, bgt	71, bgt	
7								
8	77, beq	80, bne	82, add	83, add	84, add	85, add	86, jump	

Figure 6.2 The most important loop in cccp scheduled using general speculation model.

Speculating instructions above branches moves them from less frequently executed paths to more frequently executed paths. As such, the instruction working set is increased which should result in more Icache requests and subsequently more Icache misses. The first example benchmark, cccp, is used to show the expected Icache effects. To accomplish this, the most frequently executed loop within cccp (found in the *rescan* function) was used. Based upon profile information, the IMPACT superscalar optimizer decided to unroll this loop three times. Tables 6.1 and 6.2 respectively show the scheduled code for the no speculation and general speculation models. As these tables show, none of the branches from Table 6.1 have been delayed in Table 6.2. In addition, the schedule was reduced from 20 cycles for the no speculation model to 8 cycles for the general speculation model. It should also be noted that scheduling with the no speculation model provide insufficient freedom to schedule more than 5 instructions in any cycle for the 8-issue processor.

While none of the branches in the general speculation schedule were issued later than in the no speculation schedule, the location of the branches within the Icache blocks did change as shown by Tables 6.3 and 6.4. The most notable difference is that branch instruction 59 is located in block 2 of the no speculation Icache layout while it is located in block 3 of the general speculation Icache layout. As a result, there is one additional Icache block before the branch. If branch 59 is infrequently taken, this may not increase Icache misses since both no speculation and general speculation loops are contained within only 4 Icache blocks. However, as Table 6.2 shows this branch is taken 6192 times. This means that there is an additional Icache block in the working set of the taken path of branch 59 in the general speculation schedule than in the no speculation schedule. The increased working set of this taken branch increases the chance

cccp loop		compress loop	
Branch Instruction ID	Times Taken	Branch Instruction ID	Times Taken
51	0	159	46234
53	20	164	11
59	6192	166	0
62	1753	179	39594
69	0	184	8
71	5	186	0
77	2272	199	25981
80	130	204	1
86	9182	210	26824
		214	0
Total	19554	Total	138653

Table 6.2 Branch taken frequencies. (Total corresponds to the total entrance frequency of the loop.)

Icache Block	Instruction - (instruction id, opcode)							
1								
								45, ld_c
2	48, mov	49, add	50, add	46, st_c	47, lsl	51, bgt	52, mov	53, bgt
	54, add	57, mov	55, add	58, add	56, ld_i	59, beq	60, mov	61, add
3	62, bne	66, ld_c	63, lsl	64, add	68, lsl	67, st_c	65, mov	69, bgt
	70, mov	71, bgt	72, add	75, mov	73, add	76, add	74, ld_i	77, beq
4	78, mov	79, add	80, bne	81, lsl	83, add	84, add	85, add	82, add
	86, jump							

Figure 6.3 Icache layout for cccp loop after no speculation model (16 instruction block).

of mapping conflicts with other important Icache blocks. As such, the advantages of the more aggressive schedule have resulted in greater risk of Icache misses.

Dcache Effects

Speculating load instructions above branches moves them from less frequently executed paths to more frequently executed paths. This will not only have effects on the Icache, but will also increase the frequency that the load requests are made. As such, the data working set is increased which should result in more Dcache requests and subsequently more Dcache misses. The second benchmark, compress, is used to show the expected Dcache effects. To accomplish

Icache Block	Instruction - (instruction id, opcode[* = speculative])							
1								45, ld_uc
	54, add*	57, mov*	60, mov*	52, mov*	72, add*	48, mov	49, add	55, add*
2	58, add*	61, add*	50, add	73, add*	63, lsl*	75, mov*	70, mov*	47, lsl
	46, st_c	66, ld_uc*	51, bgt	53, bgt	64, add*	76, add*	78, mov*	56, ld_i
3	79, add*	81, lsl*	68, lsl*	59, beq	62, bne	67, st_c	74, ld_i*	65, mov
	69, bgt	71, bgt	77, beq	80, bne	82, add	83, add	84, add	85, add
4	86, jump							

Figure 6.4 Icache layout for cccp loop after general speculation model (16 instruction block).

this, the most frequently executed loop within compress (found in the *compress* function) was used. Based upon profile information, the IMPACT superscalar optimizer decided to unroll this loop three times. Tables 6.5 and 6.6 respectively show the scheduled code for the no speculation and general speculation models. As the tables show, the no speculation model used 37 cycles while the general speculation model required only 18 cycles. It should also be noted that scheduling with the no speculation model provide insufficient freedom to schedule more than 6 instructions in any cycle for the 8-issue processor.

Table 6.3 shows the increased execution frequency of the six speculated loads from the general speculation schedule of this loop. By speculating a load above a particular branch, the memory reference patterns of the control flow paths reached from that branch have been altered. Depending upon the cache configuration, this could introduce more Dcache conflicts. For example, by speculating load instruction 163 above branch 159 in Table 6.6, the memory reference pattern of the paths reached by the taken path of this branch have been altered. Based upon the increased execution frequency of load number 163, and the resultant change in memory reference patterns, Dcache miss rates caused by this load could increase. Due to speculation of other loads and the change in their memory reference patterns, the total increase in Dcache misses for instruction 163 could be greater than the increase in its execution frequency.

Cycle	Instruction - (instruction id, opcode)					
1	147, sub	151, sub	156, add			
2	148, mov	157, add				
3	149, lsl	152, mov				
4	153, lsl	150, add				
5	154, xor					
6	155, lsl					
7	158, ld_i					
8						
9	159, bne	161, add	160, lsl			
10	162, add					
11	163, ld_uc2	164, bgt	165, ld_c2			
12						
13	166, beq	167, sub	171, sub	176, add		
14	168, mov	177, add				
15	169, lsl	172, mov				
16	173, lsl	170, add				
17	174, xor					
18	175, lsl					
19	178, ld_i					
20						
21	179, bne	181, add	180, lsl			
22	182, add					
23	183, ld_c2	184, bgt	185, ld_c2			
24						
25	186, beq	187, sub	191, sub	196, add		
26	188, mov	197, add				
27	189, lsl	192, mov				
28	193, lsl	190, add				
29	194, xor					
30	195, lsl					
31	198, ld_i					
32						
33	199, bne	201, add	200, lsl			
34	202, add					
35	203, ld_c2	204, bgt	205, ld_uc	209, add	208, add	207, add
36						
37	206, mov	210, bne	211, mov	212, add	213, mov	214, jump

Figure 6.5 The most important loop in compress scheduled using no speculation model.

Cycle	Instruction - (instruction id opcode[* = speculative])							
1	147 sub	151 sub	161 add*	181 add*	165 ld_uc*	167 sub*	171 sub*	156 add
2	148 mov	162 add*	182 add*	157 add	196 add*	176 add*	185 ld_uc*	187 sub*
3	149 lsl	152 mov	197 add*	177 add*	201 add*	191 sub*	205 ld_uc*	
4	153 lsl	168 mov*	150 add	202 add*				
5	154 xor	169 lsl*	172 mov*					
6	160 lsl*	155 lsl	173 lsl*	188 mov*				
7	163 ld_uc2*	158 ld_i	189 lsl*	192 mov*				
8	193 lsl*							
9	174 xor*	159 bne	170 add*	164 bgt	166 beq	206 mov*		
10	175 lsl	180 lsl*						
11	183 ld_uc2*	178 ld_i						
12								
13	194 xor*	179 bne	190 add*	184 bgt	186 beq			
14	195 lsl	200 lsl*						
15	198 ld_i	203 ld_uc2*						
16								
17	199 bne	204 bgt	207 add	209 add	208 add	210 bne		
18	211 mov	212 add	213 mov	214 jump				

Figure 6.6 The most important loop in compress scheduled using general speculation model.

Load Instruction ID	Branches Speculated Above	Change in Execution Frequency
163	159	46234
165	159, 164	46245
183	179	39594
185	159, 164, 166, 179, 184	85847
203	199	25981
205	159, 164, 166, 179, 184, 186, 199, 204	111829

Table 6.3 Increase in execution frequency of speculated loads in the compress loop after scheduling with the general speculation model.

6.2 Experimental Evaluation

This section will quantify the effects that increasing levels of scheduling freedom can have on instruction and data caches. The speculation models used in the experiments from least aggressive to most aggressive are no speculation, restricted speculation, safe speculation and general speculation. Section 6.2.1 discusses the experimental approach used to generate the cache effects. Section 6.2.2 discusses the experimental results.

6.2.1 Methodology

Compiler support for each of the speculation models has been implemented in the IMPACT-I C compiler. The IMPACT-I compiler is a prototype optimizing compiler designed to generate efficient code for VLIW and superscalar processors [7]. The benchmarks used in this study are the 14 non-numeric programs shown in Table 3.1. The benchmarks consist of 5 non-numeric programs from the SPECint92 suite and 9 other commonly used non-numeric programs. Each of the benchmarks were aggressively optimized with superblock techniques [4] and scheduled using the four speculation models varying the processor issue width from 1 to 8 instructions per cycle.

The processor model used in this study is an in-order issue superscalar processor with register interlocking. The processor is assumed to have uniform functional units, 1 branch delay slot, and the instruction set of the HP PA-RISC processor. The instruction latencies assumed are those of the HP PA-RISC 7100 (see Table 3.2). For each machine configuration, the program execution times are derived from execution driven simulations of the benchmarks in Table 3.1. During the simulations, the issue widths were varied from 1 to 8 based upon the processor model that the code was scheduled for. Dynamic branch prediction was assumed using a 1024 entry

Cache Sizes:	4K - 256K, perfect
Cache Associativity:	direct-mapped, two-set associative
Cache Block Size:	64 bytes with 12 cycle miss latency
Dcache Type:	blocking cache
Dcache Write Policy:	write-through, no write-allocate

Table 6.4 Cache configurations used in experiments.

direct mapped BTB with a 2 bit counter and a 2 cycle misprediction penalty. A perfect Dcache was used when measuring the Icache effects and a perfect Icache was used when measuring the Dcache effects. The cache configurations used for the experiments are given in Table 6.4.

6.2.2 Results

The shear volumes of data produced from the simulations made it impossible to present the individual benchmark results in this paper. In an effort to be more concise, the results presented in the subsequent figures are generated by computing the arithmetic mean of speedups for each speculation model, cache size and issue rate. Speedup was computed by dividing the execution time of the respective benchmark using the no speculation model at issue 1 with a 4K direct mapped Icache and Dcache by the execution time of the same benchmark using the specified speculation model at the specified cache size and issue rate.

6.2.2.1 Icache Performance Results

Figure 6.7 shows the performance results for direct mapped caches for the extreme speculation models - no speculation and general speculation. The first thing to observe from this figure is that the curves for the no speculation model show very little change regardless of the issue rate. In particular, there was an increase of only .35 IPC (16.9%) at issue 8 from a 4K

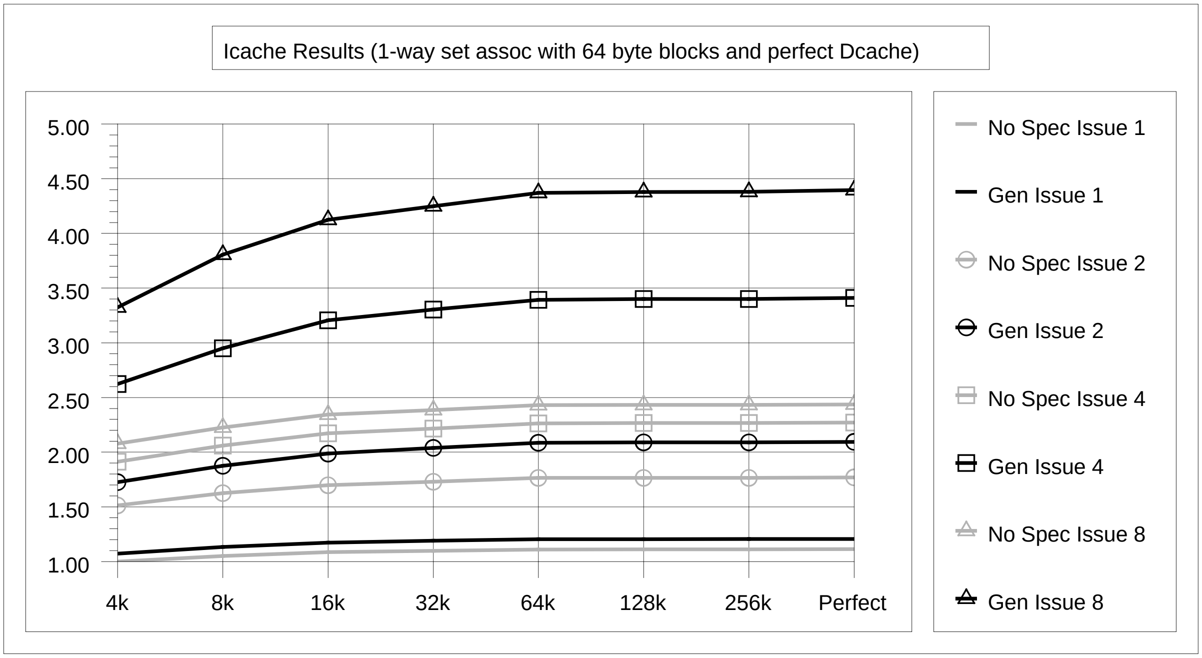


Figure 6.7 Icache effects for no speculation and general speculation models.

to a 64K Icache. In contrast, the curves for the general speculation model showed a noticeable increase from the lower issue rates to the higher issue rates. In particular, their is an increase of .36 IPC (20.9%) at issue 2, .77 IPC (29.3%) at issue 4, and 1.05 IPC (31.5%) at issue 8. Thus, the benefits from larger cache sizes are more pronounced as the issue rate increases. Finally, the performance for all speculation models stabilized with a 64K Icache.

Figure 6.8 shows the performance results for 2-way set associative caches for no speculation and general speculation models. By comparing this figure to Figure 6.7, it is clear that there is little advantage in higher associativities with Icaches larger than 8K regardless of the issue rate or speculation model. Even at the lowest cache sizes, general speculation was only able to show a 6 percent speedup at 8-issue using 2-way set associative Icaches over direct mapped Icaches.

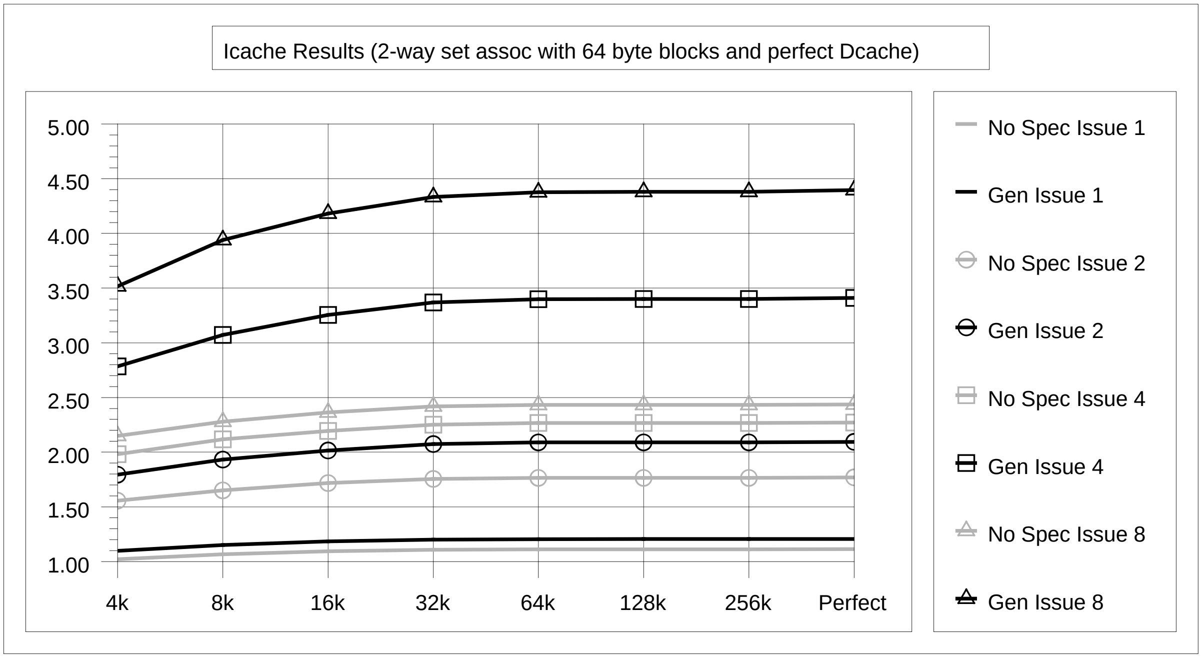


Figure 6.8 Icache effects for no speculation and general speculation models.

Figure 6.9 shows the comparative Icache results for all of the scheduling models at issue-1 and issue-8. As the figure shows, there is no significant performance advantage in using any of the aggressive speculation models for a single issue processor. Since only one instruction can be issued per cycle, the only potential slots that can be filled in the schedules of the integer benchmarks are branch and load delay slots. Therefore, there is very little opportunity to improve the performance of the benchmarks through more aggressive speculation. As a result of little speculation, only minor Icache effects are observed.

In contrast to the single issue performance, there is a clear advantage in using more aggressive speculation models at 8-issue. The no speculation model shows a 13.1 percent improvement between 4K and 64K Icaches. The restricted speculation model shows an 18.0 percent improvement, the safe speculation model shows 21.4 percent improvement and the general speculation model shows a 24.5 percent improvement over the same cache configurations. Thus, while the

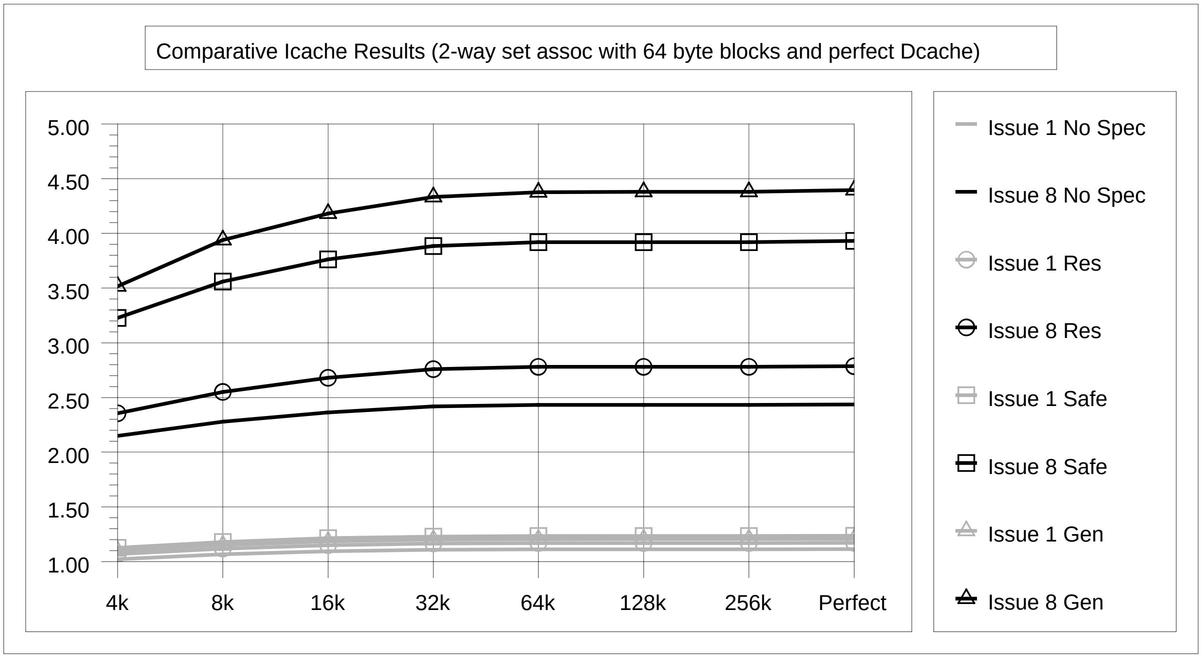


Figure 6.9 Icache effects for all speculation models at issue 1 and issue 8.

cache size was only a minor impediment to performance with lower issue rates, it is clearly a larger impediment to performance with higher issue rates for more aggressive speculation models. However, this set of benchmarks were not able to benefit from Icaches larger than 64K.

One additional point should be noted from the 8-issue results shown in Figures 6.9. The most aggressive speculation model's performance ranged from only 8.9 to 11.8 percent higher than safe speculation. Thus, safe speculation has great potential since it requires no special processor support that could potentially lead to slower clock rates. Also, it introduces none of the risks that result from ignoring scheduling errors like general speculation.

6.2.2.2 Analysis of Icache Results

To more fully understand the performance results, the Icache behavior is broken down in Tables 6.5 and 6.6. Table 6.5 contains the absolute number of read requests and read misses as well as the miss rate for each of the benchmarks in the base case. The numbers from

Benchmark	Read Requests	Read Misses	Miss Rate
008.espresso	412641852	1501759	0.36
022.li	35649513	943031	2.64
023.eqntott	1027576863	761471	0.07
026.compress	78221684	1716563	2.19
072.sc	72122569	1132815	1.57
cccp	3094004	37823	1.22
cmp	2198695	35	0.01
eqn	32813682	1508122	4.60
grep	1580207	2078	0.13
lex	46035584	208868	0.45
qsort	70546739	4041	0.01
tbl	2603306	45092	1.73
wc	1630199	35	0.01
yacc	43309632	350120	0.83
Average	130716038	586561	1.13

Table 6.5 Icache access and miss rates at issue 1 (direct mapped cache).

Table 6.6 represent the read requests and read misses as a percentage of the totals presented in the final row of Table 6.5. As Table 6.6 shows, the more aggressive speculation models tend to reduce the number of Icache read requests. This can be justified by understanding how the simulator's fetch model works. The fetch model fills buffers equivalent to twice the issue rate of the processor in an effort to provide the processor with the issue-width number of instructions at each cycle. Thus, each cycle, the fetch unit fetches a block of instructions to fill the fetch buffer. Any instructions that cannot be placed into the fetch buffer will be discarded and potentially fetched again the next cycle. Since the more aggressive speculation models have more independent instructions each cycle to choose from, the compiler is better able to group independent instructions together and reduce interlock. As such, more instructions can be issued each cycle, which reduces the need to re-fetch the same cache block repeatedly.

As Table 6.6 shows, even though the number of read requests decreased, the absolute miss rates increased for both 4K and 64K from the least aggressive speculation models to the most aggressive speculation models. In particular, there was a 1 percent increase in the miss rate

	4K Caches		64K Caches	
	Read Req	Read Misses	Read Req	Read Misses
No Speculation	0.426	0.779	0.456	0.172
Restricted Percolation	0.379	0.848	0.401	0.176
Safe Percolation	0.286	0.841	0.300	0.173
General Percolation	0.255	1.068	0.265	0.176

Table 6.6 Average icache access and miss rates at issue 8 (2-way set associative cache).

from no speculation to general speculation. There was practically no change in the Icache miss rates with 64K Icaches since the Icache was sufficiently large to hold the working set for all speculation models. While the miss rates for general speculation at 8-issue with a 64K cache is only 1.5 percent lower then the miss rate with a 4K cache, the performance was 24.5 percent higher. Thus, even a small increase in the miss rate can significantly impact the performance for the more aggressive speculation models. The impact on performance would be even more pronounced if the cache miss latency was greater than the simulated 12 cycles.

The cccp loop example shown in Tables 6.3 and 6.4 can be used to illustrate the reasons for the increase in the miss rate with the 4K Icache. Table 6.7 shows the Icache misses caused by the first instructions in each Icache blocks. The misses caused by the instruction at the start of the loop are represented with Icache block 1. There was only a negligible difference in the miss rates for the two speculation models in this block. Icache blocks 2 and 4 decreased their cache misses from the no speculation model to the general speculation model. Icache block 3 showed a significant increase in Icaches misses. Most of these misses can be attributed to migration of the misses from Icache blocks 2 and 4 to Icache block 3 due to the small 4K Icache. However, even after considering the migration of misses, there was an overall increase in misses for the loop by 16.34 percent which is attributable to the additional Icache block before the frequently taken branch number 59 in the Icache layout for the general speculation model.

Icache Block	No Speculation Misses	General Percolation Misses	General Percolation - No Speculation	Percent Change
1	2003	1980	-23	-0.01
2	3408	2486	-922	-27.05
3	2323	6952	4629	199.27
4	3227	1334	-1893	-58.66
Total	10961	12752	1791	16.34

Table 6.7 Icache misses for the no speculation and general speculation models of the cccp loop example at Issue 8 (2-way set associative, 4K Icache).

6.2.2.3 Dcache Performance Results

Figure 6.10 shows the performance results for direct mapped Dcaches for the extreme speculation models. The first thing to observe from this figure is that the curves for the no speculation model show a much smaller increase in performance than general speculation at the same issue rates. In particular, there was an increase of only .51 IPC (27.5%) at issue 8 from a 4K to a 64K Dcache while the general speculation model showed an increase of 1.32 IPC (45.4 %). In contrast to the Icache results, the performance for general speculation model still demonstrates a noticeable improvement with Dcache sizes larger than 64K.

Figure 6.11 shows the performance results for 2-way set associative Dcaches for no speculation and general speculation models. By comparing this figure to Figure 6.10, it is clear that higher associativity significantly benefits the smaller Dcaches. In particular, general speculation showed a 19 percent improvement in performance at 8-issue for a 2-way set associative 4K Dcache over a direct mapped 4K Dcache. The no speculation model showed a 14 percent improvement in performance at the same cache configurations. Both speculation models showed some performance improvement with higher associativity when using Dcaches as large as 128K. Thus, higher associativity can be better used to offset the limitations of smaller Dcaches than the smaller Icaches.

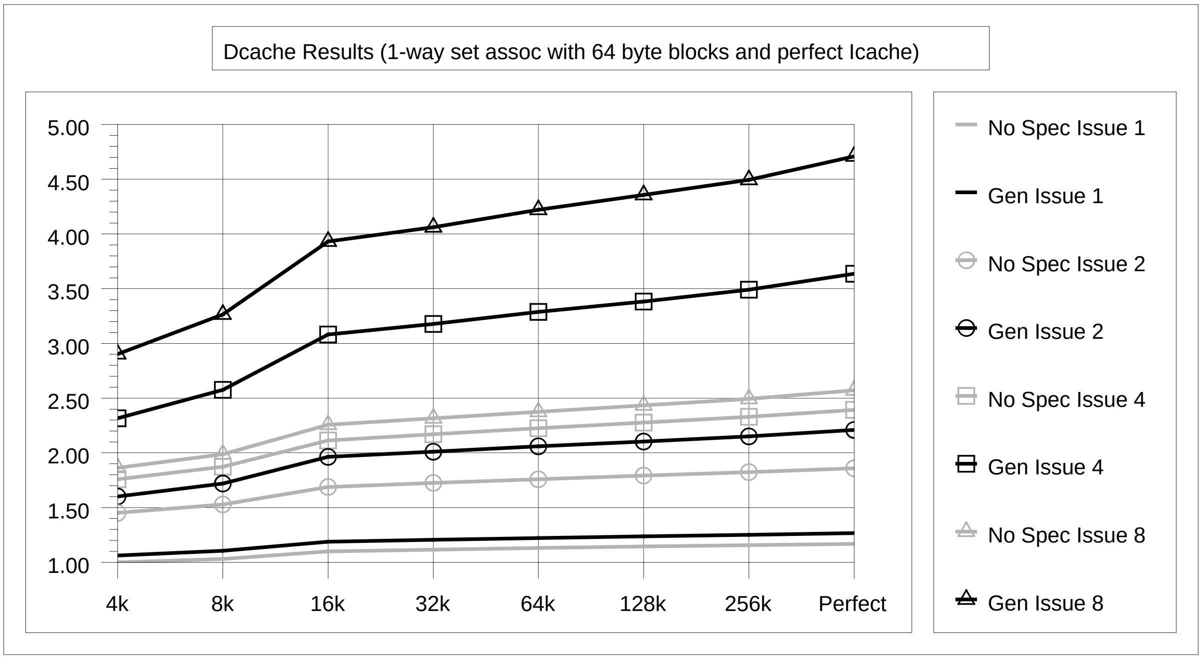


Figure 6.10 Dcache effects for no speculation and general speculation models.

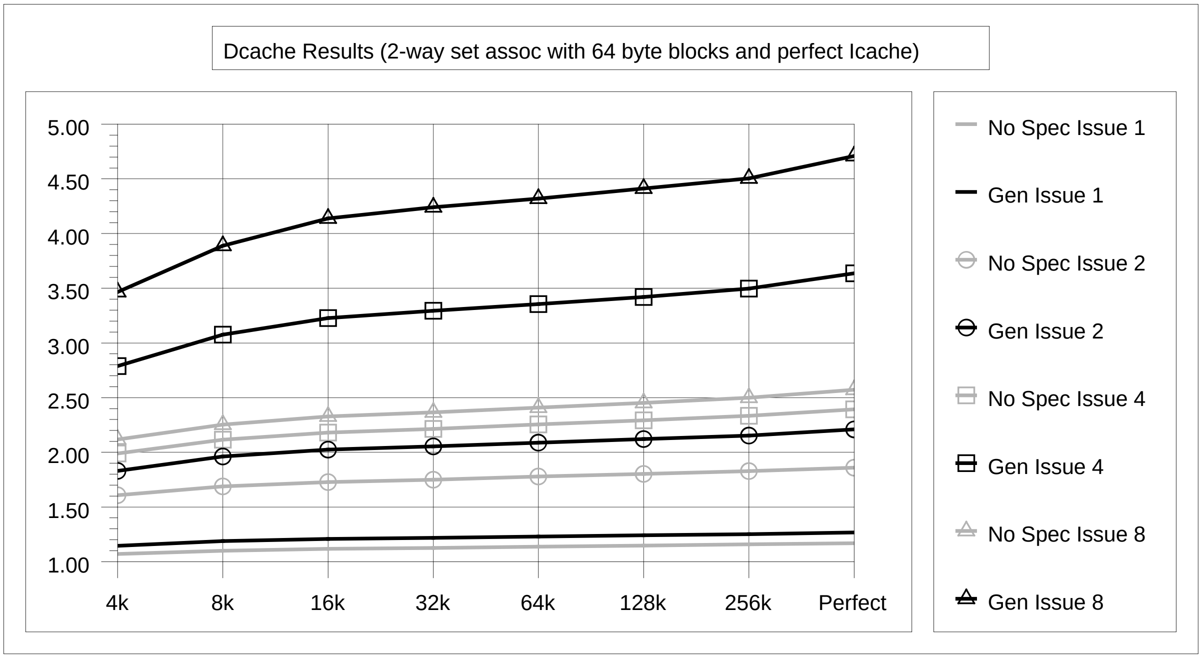


Figure 6.11 Dcache effects for no speculation and general speculation models.

Figure 6.12 shows the comparative Dcache results for all of the scheduling models at issue 1 and issue 8. As the figure shows, there is no significant performance advantage in using any of the aggressive speculation models for a single issue processor. However, at issue 8, there is a clear advantage in using the more aggressive speculation models. An increase in the Dcache size from 4K to 64K using the no speculation model resulted in a performance improvement of 13.8 percent while the restricted speculation model showed an increase of 16.3 percent. Safe speculation increased performance by 20.6 percent and general speculation increased performance by 24.6 percent over the same region. While there was no performance advantage from increasing the Icache beyond 64K, this was not the case with the Dcache. The no speculation model improved its performance to 21.5 percent higher than 4K with perfect Dcaches. Restricted speculation improved to 26.2 percent higher than 4K. Safe speculation improved to 30.1 percent higher and general speculation improved to 35.9 percent higher. Thus, small Dcaches have been shown to be a significant impediment to the potential performance of more aggressive speculation models at higher issue rates.

6.2.2.4 Analysis of Dcache Results

To more fully understand the performance results, the Dcache behavior is broken down in Tables 6.8 and 6.9. Table 6.8 contains the absolute number of read requests and read misses as well as the miss rate for each of the benchmarks in the base case. The numbers from Table 6.9 represent the read requests and read misses as a percentage of the totals presented in the final row of Table 6.8. Table 6.9 shows that the Dcache accesses increase with the more aggressive speculation models. This is caused by an increase in the working set size resulting from speculation of additional load instructions.

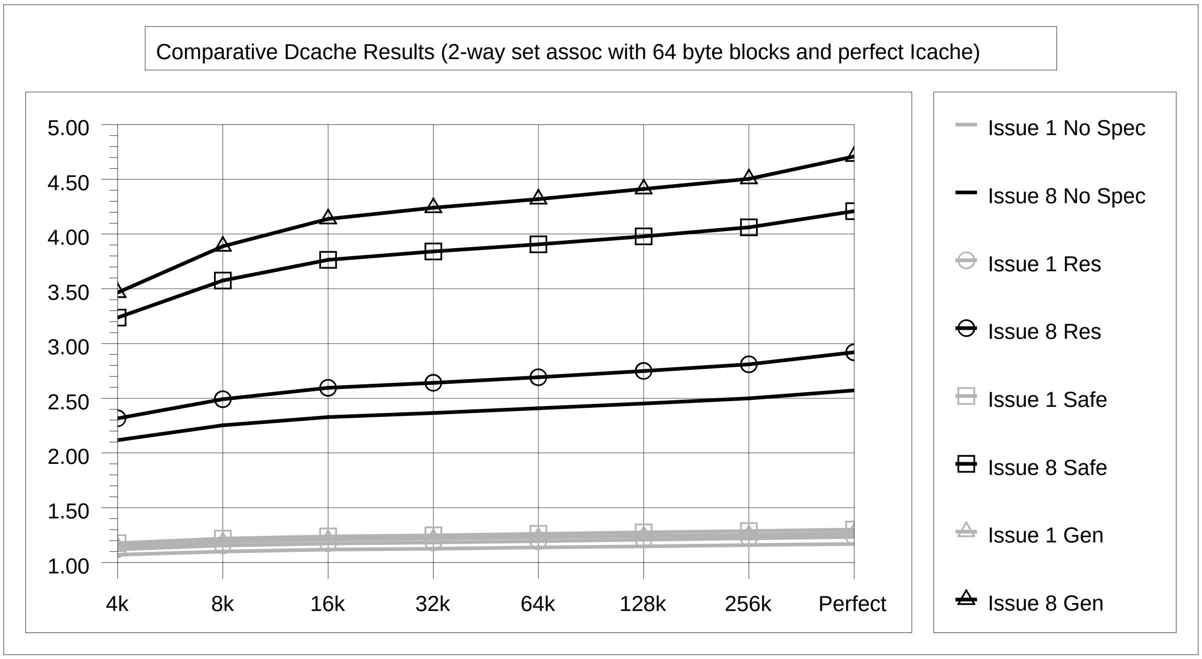


Figure 6.12 Dcache effects for all speculation models at issue 1 and issue 8.

Benchmark	Read Requests	Read Misses	Miss Rate
008.espresso	81609167	8517176	10.44
022.li	8059907	732281	9.09
023.eqntott	193278137	14396275	7.45
026.compress	9626725	3520353	36.57
072.sc	14788900	2094508	14.16
cccp	422760	21996	5.20
cmp	436248	163749	37.54
eqn	4343317	769148	17.48
grep	239327	2885	1.21
lex	7862729	314967	4.01
qsort	12206746	734301	6.02
tbl	548733	33554	6.11
wc	141527	2635	1.86
yacc	7488032	677821	9.05
Average	24382304	2284404	11.87

Table 6.8 Dcache access and miss rates at issue 1 (direct mapped cache).

	4K Caches		64K Caches	
	Read Req	Read Misses	Read Req	Read Misses
No Speculation	1.008	0.694	1.009	0.171
Restricted Percolation	1.017	0.712	1.019	0.181
Safe Percolation	1.143	0.725	1.145	0.173
General Percolation	1.313	0.776	1.315	0.193

Table 6.9 Average dcache accesses and miss rates at issue 8 (2-way set associative cache).

The decrease in the miss rate from the less aggressive to the more aggressive speculation models is miss-leading since the read requests have significantly increased. The Dcache misses actually increase from the less aggressive to the more aggressive speculation models. In particular, there was an 11.8 percent increase in the Dcache misses from the no speculation model to the general speculation model with a 4K Dcache and a 12.9 percent increase with a 64K Dcache. The 4.16 percent lower miss rate for general speculation with a 64K Dcache versus a 4K Dcache corresponds to a performance increase of 24.6 percent. Thus, the Dcache size can significantly affect the potential performance of aggressive speculation models.

The compress loop example shown in Tables 6.5 and 6.6 can be used to illustrate the reasons for the increases in Dcache misses. Table 6.10 shows the Dcache misses generated by the load instructions in the no speculation and general speculation codes based upon a 4K Dcache. It can be seen from this data that there were moderate to significant increases in Dcache misses from the no speculation case to the general speculation case. By comparing the increased Dcache miss rates for load instructions 163, 183 and 203 with their respective increases in execution frequency given in Table 6.3, it is apparent that the increase in miss rates for these loads was not constrained by the their increase in execution frequency. Other speculative loads actually caused further Dcache misses for these loads. In addition, the non-speculated load

Load Instruction	No Speculation Misses	General Percolation Misses	General Percolation - No Speculation	Percent Change
158	124951	133596	8645	6.92
163	82058	132654	50596	61.65
165	1641	2444	803	48.93
178	84617	92107	7490	8.85
183	47746	91250	43504	91.11
185	890	1685	795	89.33
198	47566	50379	2813	5.91
203	23380	49579	26199	112.06
205	501	2346	1845	368.26

Table 6.10 Dcache misses for the no speculation and general speculation models of the compress loop example at Issue 8 (2-way set associative, 4K Dcache).

instructions 158, 178 and 198 also showed an increase in Dcache misses that is attributable to other speculated loads.

6.3 Conclusions

This chapter has presented experimental results for four compiler-controlled speculation models over a variety of issue rates and cache configurations. The results indicate that the more aggressive speculation models create larger instruction and data working sets. As such, processor designers need to ensure that cache configurations can tolerate the increased working set if they expect to attain the *best* performance from aggressive speculation models. These experiments have shown that increasing the Icache and Dcache from 4K to 64K resulted in a performance increase of approximately 26 percent for the general speculation model at issue 8. Additionally, the results indicate that 2-way set associativity beneficially reduces misses for Dcaches up to 128K. In contrast, 2-way set associativity was only beneficial for Icaches up to 8K.

While small Icaches and Dcaches can significantly limit the potential performance of more aggressive speculation models, there is still an advantage in using the more aggressive speculation models at higher issue rates even if the cache configuration is held constant. Even though some of the potential advantages of the more aggressive speculation models are negated by the higher miss rates, it was not sufficient to offset the performance advantages. In particular, general speculation at issue 8 was 63.6 percent faster than no speculation with the same 4K cache configuration and issue rate. Safe speculation was 50.2 percent faster and restricted was 9.6 percent faster. When using a 64K cache, general speculation was 80 percent faster than no speculation. Safe speculation was 61.1 percent faster and restricted speculation was 14.3 percent faster. The improvements in performance were almost identical for the experiments that used a perfect Icache and varied the Dcache as those that used a perfect Dcache and varied the Icache. Thus, aggressive speculation effects the Icache and the Dcache in a similar fashion.

CHAPTER 7

CONCLUSION

7.1 Summary

Speculation of instructions above conditional branches changes the conditions under which the instruction is executed. In some cases, changing these conditions can introduce program exceptions that may result in program termination. This dissertation has presented a classification of existing speculation models based upon the support provided within the compiler and the processor to address the situation when an instruction that is desirable to speculate at compile-time can introduce an exception. The *avoid errors* class is too limited in its ability to speculate instructions since any potentially excepting instruction will not be speculated.

The *ignore errors* class assumes that any potentially excepting instructions will be replaced by non-excepting forms if they are speculated. Thus, exceptions that would have occurred when potentially excepting instructions were not speculated will not be detected if the instruction is speculated. As such, it is possible for errors to go undetected which could result in erroneous program results or even produce errors in unrelated parts of the program that will make it difficult to detect the original problem. While this model provides a great deal of scheduling freedom, it is not necessarily appropriate for use in production applications.

The *resolve errors* class of speculation models are presented as one solution to address the limitations of the avoid errors and ignore errors speculation models. These models utilize a combination of compile-time and run-time support to detect exceptions produced by speculated instructions and delay reporting the exception until it is known that the instruction would have been executed in its original un-speculated location. Additionally, mechanisms are provided in the processor that the compiler utilizes to protect the processor state.

This dissertation has presented a new speculation model that falls into the avoid errors class. The model, *write-back suppression*, eliminates the extensive processor requirements proposed by *boosting* as a means of protecting the processor state. Additionally, it eliminates the requirements for larger register files required by *sentinel scheduling* due to the requirement of extending register life times to protect the processor state. Write-back suppression utilizes the in-order retire mechanisms found in most processors to maintain the processor state. This model was shown to have speculation freedom comparable to the most aggressive ignore errors model - *general speculation*. Unfortunately, write-back suppression, along with all of the other resolve error models, requires extensions to the processor instruction set which prevent their practical usage in existing architectures.

This dissertation has presented an alternative solution to the limitations of the avoid errors and ignore errors models. This solution extends the speculation freedom of the avoid errors class by identifying potentially excepting instructions that will never cause an exception or will not cause an exception that would not have occurred in the original un-speculated location. This new speculation model, *safe speculation*, utilizes intra-procedural and inter-procedural analysis to determine the safety of potentially excepting instructions by statically propagating values information through the program. The safety analysis is performed prior to loop-level and superscalar optimizations in the back-end of the IMPACT compiler. This permits these optimizations to utilize the results of the safety analysis to perform more aggressive optimizations. It was shown that some of the aggressive optimizations such as loop unrolling can invalidate the safety analysis by changing the memory references patterns of potentially excepting load instructions. As such, the safety analysis also adds constraints to the optimization process to prevent invalidation of the results. The information produced by the safety analysis was also within the instruction scheduler to determine when control dependences are required to ensure safety of potentially excepting instructions.

Experimental results were performed to determine the benefits of this research using both intra-procedural and inter-procedural analysis. The final experimental results showed that in an 8-issue superscalar processor with perfect caches, safe speculation using intra-procedural analysis was able to attain a 20 percent speedup over the existing avoid errors model - *restricted speculation*. Safe speculation using inter-procedural analysis was able to attain a 55 percent speedup over restricted speculation and a 30 percent over safe speculation with only intra-procedural analysis. Finally, safe speculation using inter-procedural analysis was able to attain performance results that averaged within 19 percent of general speculation.

While the performance results for safe speculation were positive, several problems were identified that motivate future research. First, the absence of input files prevents beneficial analysis for some of the benchmarks such as 023.eqntott, lex and cccp. Second, in some cases it was not possible to determine the iteration space of loops. The reasons included lack of input files, loops that iterate based upon traversing linked lists and loops whose iteration space can not be determined since they are state driven. Finally, as required by the safety analysis, some loops could not be as aggressively unrolled as with general speculation in order to maintain safety of potentially excepting loads.

The final contribution to this dissertation was a study of the effects of compiler-controlled speculation on instruction and data caches. This study compared no speculation, restricted speculation, safe speculation using inter-procedural analysis and general speculation to determine if more aggressive speculation impacts performance results in the presence of caches. The study showed that even though more aggressive speculation models introduce more Icache and Dcache misses, the benefits from the greater ILP more than compensated for these misses. Thus, even with small 4K caches, it was still advantageous to use the most aggressive speculation models available.

7.2 Future Work

This dissertation has shown that safe speculation provides much more aggressive speculation freedom than restricted speculation. In several cases, its speculation freedom was shown to be the same as general speculation. However, several limitations of the current techniques prevented performance results that would eliminate the need for general speculation. Since the analysis process is dependent upon propagation of value information to potentially excepting instruction in order to determine their safety, the absence of input files can be a significant impediment to the analysis. In general, determining the safety of load instructions requires identification of all base addresses and offsets that can be performed by the load. In many cases, such as loading data from a buffer for evaluation, the contents of the buffer are not necessary. However, when the contents of the buffer are used to compute the location of a subsequent load and the contents are provide or derived from an input file, it is not possible to determine the safety of the subsequent load. This motivates further research that will permit safety analysis in the absence of input files.

Common programming practice supports modular design. Functions written to perform generic operations, such as sorts, are designed to be performed over a variety data sets. Since safety analysis must ensure that potentially excepting instructions are safe regardless of the conditions that reach these subroutines, it uses conservative analysis to maintain safety in the presence of optimizations such as loop unrolling. Using the example of the sort, assume that their are two data sets that reach the sort, one of size 256 elements and the other of size 100 elements. To ensure safety of variant loads in the sort loop given these two worst case loop iteration frequencies, one determined to be a maximum of 256 and the other a maximum of 100, unrolling would be limited to 4 times since this is the greatest common factor of both worst-case loop iterations. This limitation could prevent the necessary unrolling required to expose ILP for very wide superscalar and VLIW processors, thus preventing safe speculation from attaining the same performance as general speculation. Assuming that there was the addition of a third buffer of size 125 entries, the loop unrolling would now be prevented. One solution to removing this limitation is to implement multi-version loops. Thus, incoming loop conditions would be used to determine which optimized loop should be executed.

Due to limitations imposed by loops that iterate through linked-lists, there will always be cases that prevent safe speculation from attaining the same performance as the more dangerous general speculation model. Thus, the benefits of this model in conjunction with existing speculation models that fall into the resolve errors class should be analyzed. By identifying potentially excepting instructions that are safe, the resolve errors models are not burdened to protect as many instructions, thus reducing the run-time overhead of these models.

Finally, the safety rules presented in this research have been targeted predominantly at load instructions as well as integer divide and remainder. However, determining the safety of floating point instructions has been, for the most part, ignored until this point. Floating point instructions introduce more complications for safety analysis. In the simple case of integer divide and remainder, it is only necessary to prove that a non-zero constant reaches the divisor. However, to prove that a floating point divide will not except, we must not only test for a non-zero divisor but also ensure that underflow and overflow will not occur. If it is not possible to determine the numerator of the divide, this is not possible. Until techniques are produced that permit safe, aggressive speculation of floating point instructions, there will always be a need to use resolve error models such as write-back suppression or ignore error models such as general speculation to achieve aggressive optimization and scheduling freedom for floating point intensive programs.

REFERENCES

[1] E. M. Riseman and C. C. Foster, "The inhibition of potential parallelism by conditional jumps," *IEEE Transactions on Computers*, vol. c-21, pp. 1405–1411, December 1972.

[2] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on multiple instruction issue," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 290–302, April 1989.

[3] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 272–282, April 1989.

[4] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An effective technique for VLIW and superscalar compilation," *Journal of Supercomputing*, vol. 7, pp. 229–248, January 1993.

[5] P. Tirumalai, M. Lee, and M. Schlansker, "Parallelization of loops with exits on pipelined architectures," in *Proceedings of Supercomputing '90*, November 1990.

[6] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms.* New York, NY: McGraw-Hill, 1991.

[7] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 266–275, May 1991.

[8] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. c-30, pp. 478–490, July 1981.

[9] G. E. Haab, "Design and implementation of a data dependence analyzer for fortran programs in the impact compiler," Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1994.

[10] K. Subramanian, "Loop transformations for parallel compilers," Master's thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1993.

[11] Y. Yamada, *Data Relocation and Prefetching for Programs with Large Data Sets.* PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1994.

[12] P. Chang, *Compiler Support for Multiple Instruction Issue Architectures.* PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[13] R. A. Bringmann, "A template for code generator development using the impact-i c compiler," Master's thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1992.

[14] S. A. Mahlke, "Design and implementation of a portable global code optimizer," Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[15] S. A. Mahlke, *Exploiting Instruction Level Parallelism in the Presence of Contitional Branches.* PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1994.

[16] P. P. Chang, D. M. Lavery, and W. W. Hwu, "The importance of prepass code scheduling for superscalar and superpipelined processors," Tech. Rep. CRHC-91-18, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, May 1991.

[17] R. A. Bringmann, *Enhancing Instruction Level Parallelism Through Compiler-Controlled Speculation.* PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1994.

[18] N. J. Warter, *Modulo Scheduling with Isomorphic Control Transformations.* PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[19] R. E. Hank, "Machine independent register allocation for the IMPACT-I C compiler," Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[20] J. C. Gyllenhaal, "A machine description language for compilation," Master's thesis, Department of Electrical Engineering, University of Illinois, Urbana, IL, 1994.

[21] W. Y. Chen, "An optimizing compiler code generator: A platform for RISC performance analysis," Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[22] R. G. Ouellette, "Compiler support for SPARC architecture processors," Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1994.

[23] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL playdoh architecture specification: Version 1.0," Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, Palo Alto, CA 94303, February 1994.

[24] P. P. Chang and W. W. Hwu, "Trace selection for compiling large C application programs to microcode," in *Proceedings of the 21st International Workshop on Microprogramming and Microarchitecture*, pp. 188–198, November 1988.

[25] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," *Software Practice and Experience*, vol. 21, pp. 1301–1321, December 1991.

[26] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*, pp. 207–218, January 1981.

[27] T. Nakatani and K. Ebcioglu, "Combining as a compilation technique for VLIW architectures," in *Proceedings of the 22nd International Workshop on Microprogramming and Microarchitecture*, pp. 43–55, September 1989.

[28] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," in *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 180–192, April 1987.

[29] J. C. Dehnert and R. A. Towle, "Compiling for the cydra 5," *Journal of Supercomputing*, vol. 7, pp. 181–227, January 1993.

[30] H. Packard, *PA-RISC 1.1 Architecture and Instruction Set Reference Manual.* Cupertino, CA, 1990.

[31] D. Weaver, *SPARC-V9 Architecture Specification.* SPARC International Inc., 1992.

[32] M. D. Smith, M. S. Lam, and M. A. Horowitz, "Boosting beyond static scheduling in a superscalar processor," in *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 344–354, May 1990.

[33] M. D. Smith, M. A. Horowitz, and M. S. Lam, "Efficient superscalar performance through boosting," in *Proceedings of the Fifth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp. 248–259, October 1992.

[34] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling for superscalar and VLIW processors," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 238–247, October 1992.

[35] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling: A model for compiler-controlled speculative execution," *Transactions on Computer Systems*, vol. 11, November 1993.

[36] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach.* San Mateo, CA: Morgan-Kaufmann, 1990.

[37] J. A. Fisher, "Global code generation for instruction-level parallelism: Trache scheduling-2," Tech. Rep. HPL-93-43, Hewlett Packard Computer Research Center, June 1993.

[38] T. Ball and J. R. Larus, "Branch prediction for free," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pp. 300–313, June 1993.

[39] R. E. Hank, S. A. Mahlke, R. A. Bringmann, J. C. Gyllenhaal, and W. W. Hwu, "Superblock formation using static program analysis," in *Proceedings of the 26th Annual International Symposium on Microarchitecture*, December 1993.

[40] J. A. Fisher and S. M. Freudenberger, “Predicting conditional branch directions from previous runs of a program,” in *Proceedings of the 5rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 85–95, October 1992.

[41] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.

[42] K. Kennedy, “Safety of code motion,” *Journal of Computer Mathematics*, vol. 3, pp. 117–130, 1972.

[43] R. M. Stallman, *Using and porting GNU CC.* Free Software Foundation, Inc., 1989.

[44] G. Kane, *MIPS R2000 RISC Architecture.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987.

[45] D. Bernstein and M. Rodeh, “Proving safety of speculative load instructions at compile time,” in *Fourth European Symposium on Programming*, 1992.

[46] U. Mahadevan and S. Ramakrishnan, “Instruction scheduling over regions: A framework for scheduling across basic blocks,” tech. rep., Californial Language Laboratory, Hewlett Packard, Cupertino, CA, 1994.

[47] *American National Standard: Programming Language FORTRAN.* ANSI X3.9-1978, April 1978.

[48] B. Kernighan and D. Ritchie, *The C Programming Language.* Englewood Cliffs, NJ: Pretice Hall, 1988.

VITA

Roger Alexander Bringmann was born in Tuscaloosa, Alabama in 1962. He pursued his undergraduate studies at the University of South Alabama in Mobile, Alabama where he received his B.S. degree in Computer Science in 1983. After receiving his B.S., he joined QMS, Inc. in Research and Development where he advanced to the position of Senior Member of Technical Staff. In the fall of 1990, he began his graduate studies in Computer Science at the University of Illinois in Urbana, Illinois. He completed his M.S. degree in Computer Science in 1992. During his tenure at the University of Illinois, he has been a member the IMPACT project directed by Professor Wen-mei W. Hwu. After completing his Ph.D., he will return to QMS, Inc. to assume the position of Executive Director of Advanced Technology.