Sentinel Scheduling with Recovery Blocks

David I. August Brian L. Deitrich Scott A. Mahlke

Center for Reliable and High-Performance Computing
University of Illinois
Urbana, IL 61801

January 31, 1995

Abstract

This paper introduces a method for speculative execution, sentinel scheduling with recovery blocks, which is an improvement upon sentinel scheduling. This method is compared against speculative models that have been proposed previously. The hardware and compiler support needed for sentinel scheduling with recovery blocks is fully described. Quantitative results of speedup and code growth are presented for 14 benchmark programs. This method was fully implemented with the IMPACT compiler. Sentinel scheduling with recovery blocks is shown to have a low hardware cost, while achieving good performance results.

Index terms - speculative execution, sentinel scheduling, computer architecture, compilers, static scheduling, recovery blocks, instruction-level parallelism, potentially excepting instructions

1 Introduction

Instruction-level parallelism (ILP) is necessary in order to fully utilize the functional units in wide issue superscalar and VLIW architectures. There is insufficient ILP within basic blocks, especially for non-numeric applications, to fully utilize the processor resources that are available. [1] [2] [3] Techniques such as software pipelining [4] [5] [6] and predicated execution - used in conjunction with software pipeline loop scheduling [7] or straight-line code scheduling [8] - are effective for exposing ILP only when branch conditions can be exposed in advance. For applications where accurate branch prediction is not possible, speculative execution is an important source of ILP. [9] [10] [3] Lack of ILP is intimately tied to the increasingly important problem of coping with high memory latency. As such, speculation can also diminish the negative effects of memory latency.

Speculative execution refers to the execution of an instruction before it is known whether the instruction needs to be executed. During compilation, the instruction becomes speculated when it is moved above one or more conditional branches. Code speculation introduces some problems that could effect proper code execution. When an instruction is speculated, it can be executed more times than if it were not speculated. These extra executions must have no effect on the outcome of the application. This includes ignoring exceptions during the extra executions of potentially excepting instructions (PEI's).

Several architectural and compiler models have been proposed to deal with the problems of speculative execution. This paper presents a background of the most recognized models to date. Then, it introduces a new model, sentinel scheduling with recovery blocks (SSRB), which is an improvement upon sentinel scheduling, a scheme with many positive aspects. Throughout, the advantages of SSRB over existing speculation models will be presented.

2 Background and Related Work

This section presents a number of speculation models. In turn, each of these model is presented and evaluated. The first two models, restricted and general speculation, are on opposite extremes with respect to dealing with PEI's. This fact makes them a good basis of comparison for generating relative performance results. The remaining methods, instruction boosting, write-back suppression, and sentinel scheduling, are used for comparison with SSRB in areas such as hardware overhead, speculation distance, execution time, and register pressure.

2.1 Restricted Speculation Model

One way to deal with PEI's is to simply not speculate them. Restricted speculation is a simple speculation model that does just this. [3] The compiler only speculates instructions that it can guarantee will not except. This means that loads, stores, integer divides, and all floating point operations as well as any other PEI cannot be speculated. Since speculation is limited only to non PEI's, there are fewer opportunities for ILP. Additionally, since no memory operations are speculated, this model has a reduced benefit with respect to hiding memory latency. Therefore, restricted speculation is used as a lower bound base case for our performance analysis.

2.2 General Speculation Model

General speculation's solution to proper exception handling is to provide a non-excepting version of each PEI which is used when it is speculated. [3] The hardware ignores all exceptions, except for page faults and TLB misses, during execution of a speculated PEI. This model is easy to implement and allows the maximum performance to be obtained with minimal costs to the architecture. A major drawback is that exceptions which were caught by the original program may not occur in the speculated version of the same code. This allows the program to continue execution despite incorrect execution of the speculated instruction. Many times, a program will terminate normally, yet produce incorrect results that may go undetected. The general speculation model's results are used as an upper bound base case for our analysis since it is optimal in terms of performance.

2.3 Instruction Boosting Scheduling Model

Instruction boosting proposed by Smith, et al. combines extra hardware support in the form of shadow register files and extra compiler support by generating recovery blocks to handle exception recovery. [9] [11] When an exception occurs for a speculated PEI, the exception is recorded with respect to one of the shadow register files. If no exception occurs for a speculated PEI, the results of the speculated instruction are put into the shadow register file. At the commit point for a speculated PEI, the shadow register is examined to see if an exception has occurred. If there was no exception, the result located in that shadow register is moved to its corresponding normal register. If there was an exception, the shadow register is ignored, and the processor re-executes instructions in an appropriate recovery block. The compiler generated recovery blocks regenerate the exception as if the code had not been speculated.

This model provides accurate detection and handling of speculated PEI's. However, there is a substantial hardware cost. The scheme requires multiple copies of the register file, a counter for each register in each register file, and a field in each speculated instruction indicating the number of branches it was moved above. In fact, it needs as many register files and counter files as the maximum number of branches any single instruction is speculated above. Given that some integer benchmarks have an average of 2-4 instructions between each branch, this method either has extremely high cost or very low performance. [12] The proponents of instruction boosting acknowledge this limitation and suggest the use of hybrid models. However, the only reasonable solution still requires one extra register file, a counter for each register, a few control bits, and a field in each speculated instruction. The size of the field determines how many branches a speculated instruction can be moved above.

Instruction boosting cannot speculate spilled registers since the counter needs to be resident. Not being able to speculate spilled registers can restrict speculation of instructions when register pressure is significant. This nullifies some of the advantages that instruction boosting's large hardware overhead provided.

When using a reasonable amount of hardware, instruction boosting can only support one path of execution between a speculated instruction and its original location. The severity of this limitation is discussed in more detail in the section 3.

2.4 Write-Back Suppression Model

Write-back suppression delays the exception of a speculated PEI. After encountering the speculated exception, it systematically suppresses all updates to the register file by other speculated instructions that were located after the speculated exception in the original program. [13] These instructions are identified by a field in each instruction indicating how many basic blocks above its home block it was speculated. A check instruction located in the PEI's home block indicates that recovery should occur. If there was a PEI from that basic block that excepted, the exception handling is performed upon reaching the check instruction. Exception handling consists of re-executing the excepting PEI and all subsequent instructions that were suppressed in the original pass.

Write-back suppression, like instruction boosting, provides accurate recovery from excepting speculated PEI's. A drawback of write-back suppression is that it requires an extra k-bit (where an instruction is allowed to move above 2^{k-1} branches) field on every instruction in order to determine which home block it came from.

A disadvantage of write-back suppression as presented is that, like instruction boosting, it can only handle one path between the speculated instruction and its origin. Another disadvantage of write-back suppression is that it cannot spill any registers associated with a speculated instruction. This limits the speculation potential when the register pressure is high. Write-back suppression also has a hardware overhead due to the fact that it has to support a stack of suppressed instruction PC's which is used during the recovery process.

2.5 Sentinel Scheduling Model

A model which requires much less hardware to handle excepting speculated instructions correctly is sentinel scheduling. [14] Sentinel scheduling is a compiler based technique that requires relatively few changes to the processor architecture. When an exception occurs for a speculated PEI, the destination register of the instruction is marked as excepting. The program counter of the excepting instruction is then written into the destination register itself. The previous value in this register is not needed since it was going to be overwritten by the excepting instruction anyway. Subsequent speculative instructions which use the result of that excepting speculative instruction are suppressed and instead propagate the PC and exception tag to their destination register. Source operands for speculative instructions are preserved by ensuring that the scheduler and register allocator do not allow any instructions to overwrite a speculative instruction's source operands until the exception is recovered or avoided. Recovery is initiated by a sentinel which may be an explicit or, more commonly, an implicit check. An explicit check is represented by a special additional instruction created solely to act as a sentinel. Its register operands are checked for an exception tag. An implicit check is a normal non-speculated instruction whose source registers are checked for an exception flag before it is executed. The actual process of recovery is started by setting the PC to the value found in the register and re-executing all speculative instructions until the check instruction that initiated the recovery is reached.

The main benefit of sentinel scheduling is its low hardware overhead. Only the extra S bit in the instruction and an extra E bit in every register are needed. The S bit indicates if an instruction is speculated or not. The E bit is the exception flag discussed previously.

The only significant flaws with sentinel scheduling are due to the complexity involved with inline recovery. First, the scheduler and register allocator are inhibited by the requirement that all source and destination operands of speculated instructions cannot be reused until after it is known that those values are not needed for recovery. Second, the source and destination registers of speculated instructions cannot be spilled easily. This is due to the fact that inline recovery cannot automatically load spilled registers during recovery. In the paper that introduced sentinel scheduling, the authors dealt with this problem by despeculating any instruction which would have needed spill code. [14] Third, since all speculated instructions are re-executed, their re-execution cannot have any adverse effects. For example, a self-antidependent (e.g., RA = RA + 1 is self-antidependent) instruction cannot be speculated unless it is broken up into an instruction that is not self-antidependent and an added renaming instruction which is added to the home block. Finally, inline recovery needs to know the direction of all branches between the initial speculated instruction and the check, so that the control flow is reproduced accurately. This is not a trivial problem, unless only one path of execution is used, as in the case of superblocks. How this problem is dealt with by sentinel scheduling has not been addressed in previous papers about sentinel scheduling. These and other negative effects due to inline exception recovery with sentinel scheduling led to the idea of using recovery blocks. The remainder of this paper discusses the use of recovery blocks in doing exception recovery with sentinel scheduling.

3 Sentinel Scheduling with Recovery Blocks

Sentinel scheduling with recovery blocks uses sentinels and exception tags on registers to know when to recover, just as traditional sentinel scheduling. However, unlike traditional sentinel scheduling, SSRB goes to a special code sequence, called a recovery block, to handle the exception. A recovery block is a compiler generated block of code used for the sole purpose of recovery. Performing recovery in special recovery blocks does increase code size, but instruction cache effects are minimized by locating them away from the frequently executed program code.

In traditional sentinel scheduling, a speculated instruction that excepts writes its program counter into the destination register. This PC value is not needed in SSRB since inline recovery is not performed. Instead, each basic block with a sentinel in it is assigned its own recovery block. The processor calculates the address of a basic block's recovery block based upon a compiler generated lookup table. The lookup table is referenced by the address of the first instruction in the basic block and it returns the address of the corresponding recovery block. In SSRB, the destination register of a speculated instruction can be used for other purposes.

Since traditional sentinel scheduling re-executes all speculated instructions during recovery, all registers that are involved must contain the same value they had during the initial execution sequence. This is done by extending the interval in which the source registers retain the same value. The compiler makes sure that these registers have their live ranges extended, don't get redefined, and are not spilled. Register pressure may actually force code to be despeculated. To alleviate this problem, SSRB uses the destination operand of the PEI to hold the value of the first source operand. This frees the first source operand for immediate reuse. Any other source operands in the PEI cannot be reused. Reuse of one register is not significant. However, in many cases, there are many instructions which are dependent on the excepting PEI. Since these instructions are also suppressed, their first source operand is copied into their destination operand. The combined effect of one register freed for reuse per suppressed instruction is significant. Since reusing one register may have a greater positive impact than reusing another, it is suggested that special instructions be added that allow reordering of source operands when their order is significant with respect to the operation.

Sentinel scheduling with recovery blocks can support multiple execution paths from a speculated instruction to the sentinel. Unlike sentinel scheduling, SSRB does not need to know which path to re-execute because recovery is performed using recovery blocks. This is also unlike instruction boosting and write-back suppression since SSRB does not depend on a predefined execution path as the mechanism to keep track of the excepting speculated PEI's.

Instruction boosting, write-back suppression, and sentinel scheduling cannot handle the case shown in the original code found in Figure 1 without a technique like superblock formation. It is not possible for these methods to speculate an instruction from Basic Block (BB) 5 into BB 1 since there are two paths between BB 1 and BB 5. These methods must choose one path, or trace, in which instructions can be speculated. Superblock formation is used by these methods to create this path. Sentinel scheduling with recovery blocks is not limited to one path and does not necessarily

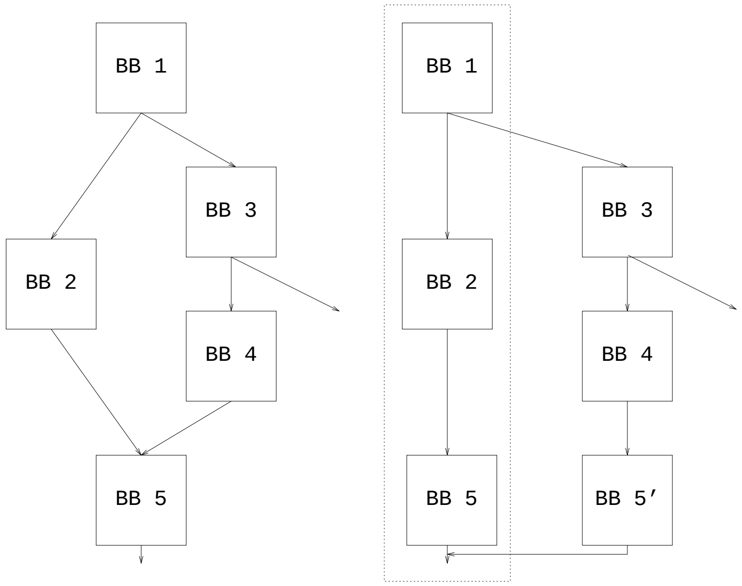

Figure 1: Superblock formation for a simple if-then-else structure.

need to perform superblock formation.

The details of the architecture support needed for SSRB is discussed in 3.1 and the details of the the compiler support required is discussed in 3.2.

3.1 Architectural Support

The most obvious extension to the micro-architecture is the addition of a bit in the opcode field of every instruction to indicate whether the instruction is speculated or not. This bit is referred to as the S bit. The compiler must set this bit if it chooses to speculate a particular instruction. Another extension is an exception tag that is added to each real register in the register file. This tag, or E bit, is used to keep track of exceptions and recovery.

The complete significance of the E and S bits becomes clear when the effects they have on the normal operation of the processor are detailed. In a traditional processor, an instruction executes unless it causes an exception, in which case the exception is handled. In a processor with SSRB support, the E bit, the S bit, and whether an instruction excepts are combined to determine what action should be taken. A summary of the actions needed with respect to the execution of each

S bit	$\cup src(*).E\_bit$	*excepts*	$dest.E\_bit$	$dest.data$	*action*
0	0	0	0	instruction result	none
0	0	1	0	-	take exception.
0	1	0	0	-	execute recovery block.
0	1	1	0	-	execute recovery block, then take exception.
1	0	0	0	instruction result	none
1	0	1	1	$src(1).data$	none
1	1	0	1	$src(1).data$	none
1	1	1	1	$src(1).data$	none

Table 1: Instruction action taken by sentinel scheduling with recovery blocks.

instruction is presented in Table 1.

If an instruction, whether it is speculated or not, executes normally, its results are placed in the destination register. In addition, the instruction also clears the destination register's E bit.

If a speculated instruction would except, or if one of its source operand registers E bits are set, special action is taken. Instead of executing, it sets the E bit of its destination register and copies its first source operand register into the destination operand register. For instructions where it is impossible to change the operand order without changing the instructions meaning (e.g., division, subtraction, etc.), multiple versions of these instructions are added to the instruction set so that the compiler can pick which register to save. For example, a divide instruction could be developed which overrides the hardware default of always saving the first source operand register. Another option is for a divide instruction that puts the divisor first to be complimented with one that puts the divisor second.

If a non-speculated instruction has one of its source operand registers E bits set, that instruction acts as a sentinel, and the processor begins execution of the appropriate recovery block. This instruction may be an implicit check or an explicit check. The explicit check is an additional instruction whose sole purpose is to check the E bit of all its source registers. It is used for cases where no implicit check is available. Upon returning from the recovery block, the instruction is executed. If the instruction excepts, the exception is signaled immediately. In all these cases, the E bit of the destination operand is also cleared.

When exception processing is initiated, the hardware determines the location of the appropriate recovery block with the use of a compiler generated lookup table. The address of the first instruction in the basic block is used as the index into this table which contains the address of the appropriate recovery block. This is similar to the method that was utilized in Smith's instruction boosting model. [11] The hardware also records the current PC on the stack, so that upon completion of the recovery block, execution can continue where it left off.

Execution in the recovery block is performed in a slightly different manner than program code. Most every instruction in the recovery block is self-antidependent. The reason for this will become clear in Section 3.2. Since executing self-antidependent instructions extra times is destructive, the hardware must execute only the relevant instructions in the recovery block. This is done by only executing instructions whose destination has the E bit set. Since a register can only be a destination once in any particular recovery block, the E bit may be reset immediately after the instruction is executed. Upon completion all exceptions related to a home block are recovered by a single pass through the recovery block.

As mentioned previously, the exception bit must be maintained in the same manner as the register contents. For this reason, modifications must be made to how the processor performs context switches so that E bits are saved and restored. This also means that special spill code loads and stores must be created for the same purpose. Another reason a special spill code store must be created is to ensure that spill code does not act as a sentinel.

3.2 Compiler Support

The compiler is responsible for ensuring that a sentinel is present for every speculated PEI in its home block, generating the recovery blocks, and removing self-antidependencies for registers which need to have values maintained to the recovery block.

An important concept to understand is a PEI's flow dependence chain. A PEI's flow dependence chain is the chain of instructions which are flow dependent on the PEI, or are flow dependent on an instruction that is part of the PEI's flow dependence chain. For example, a PEI's flow dependence chain could consist of N instructions and only one of them is flow dependent on the PEI. The first instruction is flow dependent on the PEI, the second instruction is flow dependent on the first instruction, the third instruction is flow dependent on the second instruction, and so forth. The flow dependence chain only contains speculated instructions. The flow dependence chain stops at a sentinel.

The flow dependence chain determines if an explicit check is needed or not. An instruction can act as an implicit check for a PEI if it is flow dependent on an instruction located in the PEI's flow dependence chain, belongs to the same home block as the PEI, and is not speculated. If there is no instruction in a PEI's home block that meets this criteria, an explicit check must get added.

The instructions put into the recovery block consist of all the PEI's from a home block, and all the speculated instructions found in the PEI's flow dependent chains. At the end of the recovery block, a return operation is inserted so the processor knows that exception recovery is complete. This return instruction lets the program execution restart at the sentinel instruction that initiated the recovery.

Speculative instructions that were originally located in the home block where the recovery block is being built are non-speculative in the recovery block. If the recovery block is being executed, program control has already ensured that these instructions should execute. Speculative instructions from other home blocks should remain speculative in the recovery block because it is not known yet whether program control will allow these instructions to execute.

There is only one recovery block per home block. This allows the hashing scheme (the lookup table discussed in the previous section) to work easier, and limits the size of some recovery blocks. Several PEI's may be from the same home block, and they may share common instructions in their flow dependence chain. These common instructions appear only once in a common recovery block, but would appear multiple times if each check had its own recovery block.

Sentinels are not required to be located next to one another. Therefore, it is possible that an instruction speculated from a different home block could be located between sentinels, and still be part of a PEI's flow dependence chain. This may cause the speculated instruction to be executed more than once, but this does not hurt anything as long as the instruction is not self-antidependent.

If there is a self-antidependence, it needs to be broken up.

Source operands of instructions present in the recovery blocks are altered to reflect the fact that the hardware automatically stores the first source register operand's value into the destination register when a speculated instruction excepts. The only time the first source operand is not altered for the recovery block is when all the source operands for an instruction are also destination operands for other instructions in the recovery block. Note that this creates many self-antidependent instruction. Therefore, the recovery block must only execute once to recover for all instructions originally in a home block.

Self-antidependencies are an issue if a source register's value must be maintained from a speculated instruction to its recovery block. This is solved with register renaming. A new register is used in place of the destination register of a self-antidependent instruction. Then, all references to the old register from that point until an appropriate sentinel are renamed to the new register. After the sentinel, the value in the new register is moved back to the old register if necessary. The number of times that this needs to be done can be minimized by having the compiler choose the source registers that have self-antidependent instructions to be saved automatically in the destination register.

Self-antidependencies are allowed for destination registers that are already going to be part of a recovery block. This is allowed as long as the source register to be automatically saved is the same as the destination register and the speculated self-antidependent instruction always excepts when entry into the recovery block is going to occur (a self-antidependent instruction cannot be allowed to execute more than once). If these requirements cannot be met, the self-antidependence solution described in the previous paragraph must be used.

3.3 Code Example

Table 2 shows how an example code segment would get scheduled using SSRB. The home block seen in the original code sequence is defined by the two branches (instructions 1 and 7). The scheduled code has speculated four instructions (instructions 2, 3, 4, and 5). Instruction 6 cannot be speculated since it is a store instruction. Two PEI's have been speculated from the original

1	$beq\ r5, 1, 50$	2	$\langle S\rangle\ r10 \leftarrow mem(r11)$	$2'$	$r10 \leftarrow mem(r10)$
2	$r10 \leftarrow mem(r11)$	3	$\langle S\rangle\ r3 \leftarrow r2/r1$	$3'$	$r3 \leftarrow r3/r1$
3	$r3 \leftarrow r2/r1$	4	$\langle S\rangle\ r6 \leftarrow r3 + r4$	$4'$	$r6 \leftarrow r6 + r3$
4	$r6 \leftarrow r3 + r4$	5	$\langle S\rangle\ r9 \leftarrow r10 + r6$	$5'$	$r9 \leftarrow r10 + r6$
5	$r9 \leftarrow r10 + r6$	1	$beq\ r5, 1, 50$	$6'$	$return$
6	$mem(r7) \leftarrow r6$	6	$mem(r7) \leftarrow r6$		
7	$bne\ r10, 0, 100$	8	$check\ r10$		
		7	$bne\ r6, 0, 100$		
A.	Original Code Sequence	B.	Scheduled Code Sequence	C.	Recovery Block

Table 2: Example of code scheduled for sentinel scheduling with recovery blocks.

home block (instructions 2 and 3). Instruction 5 is part of instruction 2's flow dependence chain; instructions 4 and 5 are part of instruction 3's flow dependence chain.

Instruction 6 acts as a sentinel for instruction 3 since it has not been speculated, belongs to the same original home block as instruction 3, and is flow dependent on instruction 4 which is a member of instruction 3's flow dependence chain. Since instruction 4 is not a member of instruction 2's flow dependence chain, instruction 6 cannot act as a sentinel for instruction 2.

This means that instruction 2 needs to have an explicit check for it in the home block. The check (instruction 8) determines if instruction 2 excepted. The explicit check could have had a value other than r10. It could have been r9. If r9 had been chosen instead, the check could have acted as a sentinel of instruction 2 and 3. An implicit check does not add any extra code, and if it was scheduled before the explicit check of r9, the implicit check would still force entry into the recovery block.

The recovery block for this home block is also shown. The key thing to note is the use of the destination register in saving a source register. This means that the only registers that need to have their live ranges extended into the home block are r10, r3, r1, and r6. The scheduler and register allocator would be free to reuse r11, r2, and r4 as long as their live-ranges have expired. In the original sentinel scheduling model, these live-ranges would have been extended to the sentinel. Also note that instruction $5'$ of the recovery block does not use the destination register as a source register. This happens because both of its original source registers are defined by previous instructions in the recovery block.

Category	*Boosting*	*WBS*	*Sentinel Scheduling*	*SSRB*
Addition To Instructions	k-bit field	k-bit field	1-bit field	1-bit field
Hardware Per Register	Counters and Registers	None	1-bit exception flag	1-bit exception flag
Additional Hardware	Jump Table	Stack and Counter	Instructions	Jump Table and Instr.
Spill Speculated Registers	N/A	No	No	Yes
Maximum Speculation	very limited	limited	unlimited	unlimited
Multiple Execution Paths	no	no	extra hardware	yes
Code Size	Recovery Blocks	minimal	minimal	Recovery Blocks
Performance	high	high	high	high

Table 3: Relative merits of safe speculative execution models.

4 Evaluation of Complete Recovery Methods

The relative merits of the speculative models, described earlier in this paper, are shown in Table 3. The four models, Instruction Boosting, Write Back Suppression (WBS), Sentinel Scheduling, and SSRB, are compared.

SSRB and Sentinel Scheduling require the fewest number of additional bits in each instruction. They only require one additional bit to every instruction as opposed to a field. SSRB does require hardware for a jump table and extra instructions, but overall it requires much less hardware than instruction boosting. Sentinel Scheduling and write-back suppress also require special hardware, but the hardware they require is also much less than the hardware involved with instruction boosting.

SSRB also has more scheduling freedom than the other models because it allows any speculated register to become spilled. Instruction boosting, by definition, does not have to spill any speculated registers, but it does so at a large hardware cost and cannot speculate as much in the presence of high register pressure. SSRB also has no limit on the number of branches that an instruction can be speculated above, and can support multiple execution paths.

SSRB, like Instruction Boosting, has more code growth than the other two models because compiler generated recovery blocks are used. Since the exceptions that are being delayed should occur infrequently, the recovery blocks should rarely need to be used. Where a PEI is intentionally allowed to except, the PEI should not be allowed to speculate. In these cases, performance is degraded too much to allow an instruction that is known to except to be speculated using SSRB.

The performance of these models is comparable. SSRB may get a slight edge over the other models because it provides more scheduling freedom. In any case, the important thing about SSRB is that it provides good performance at a smaller cost than previously presented speculative models.

5 Experimental Evaluation

In section 5.1, the details of our implementation are discussed Then the speedup obtained by sentinel scheduling with recovery blocks is analyzed in section 5.2. Finally, the amount of code growth is presented and discussed in section 5.3.

5.1 Implementation

The best way to evaluate the effectiveness of sentinel scheduling with recovery blocks is to use a compiler to generate code for a machine with such support. The compiler generated code can be used to evaluate code growth associated with the recovery blocks, explicit checks, and other necessary code changes. The generated code can also be used to analyze the performance of a machine with SSRB support.

For this project, support for sentinel scheduling with recovery blocks was added to the instruction scheduler of the IMPACT-I compiler. The IMPACT-I compiler is a prototype optimizing compiler designed to generate efficient code for VLIW and superscalar processors. [3] While SSRB can support multiple traces, this implementation limited speculation to within a single superblock.

Using the IMPACT-I compiler with SSRB support, we generated code for 14 non-numeric benchmarks. These benchmarks are shown in Table 4. The benchmarks consist of 5 programs from the SPECint92 suite and 9 other commonly used non-numeric programs.

The instruction set used is based upon the HP PA-RISC instruction set with extensions to support SSRB. [15] Instruction latencies of the HP PA-RISC 7100 are assumed. Table 5 shows

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

Table 4: Benchmarks

these assumed instruction latencies. The basic processor has 64 integer registers, 64 single precision floating point registers which can accommodate 32 double precision values, and an 8 entry store buffer. An issue rate of 8 is also assumed with at most 1 branch per cycle. Program execution time is derived using profile based calculation of the worst case cycle count given a 100% cache hit rate and static branch prediction. Branch mispredictions incur a 2 cycle penalty.

Function	Latency	Function	Latency
Int ALU	1	FP ALU	2
memory load	2	FP multiply	2
memory store	1	FP divide(SGL)	8
branch	1 / 1 slot	FP divide(DBL)	15

Table 5: Instruction latencies.

5.2 Speedup

Given that the general speculation model yields the best possible result for any given speculative scheduler, a good measure of any speculation model's effectiveness is the percentage of general's performance it can attain. Justification for using an exception recovery model at all is how much performance the model gives over restricted speculation alone. Sentinel scheduling with recovery block's performance compared to no speculation, restricted speculation, and general speculation is presented in Figure 2. Performance is relative to the no speculation case which is normalized to 1. It should be mentioned that due to IMPACT-I's other optimizations, speedup with respect to unoptimized code is much greater.

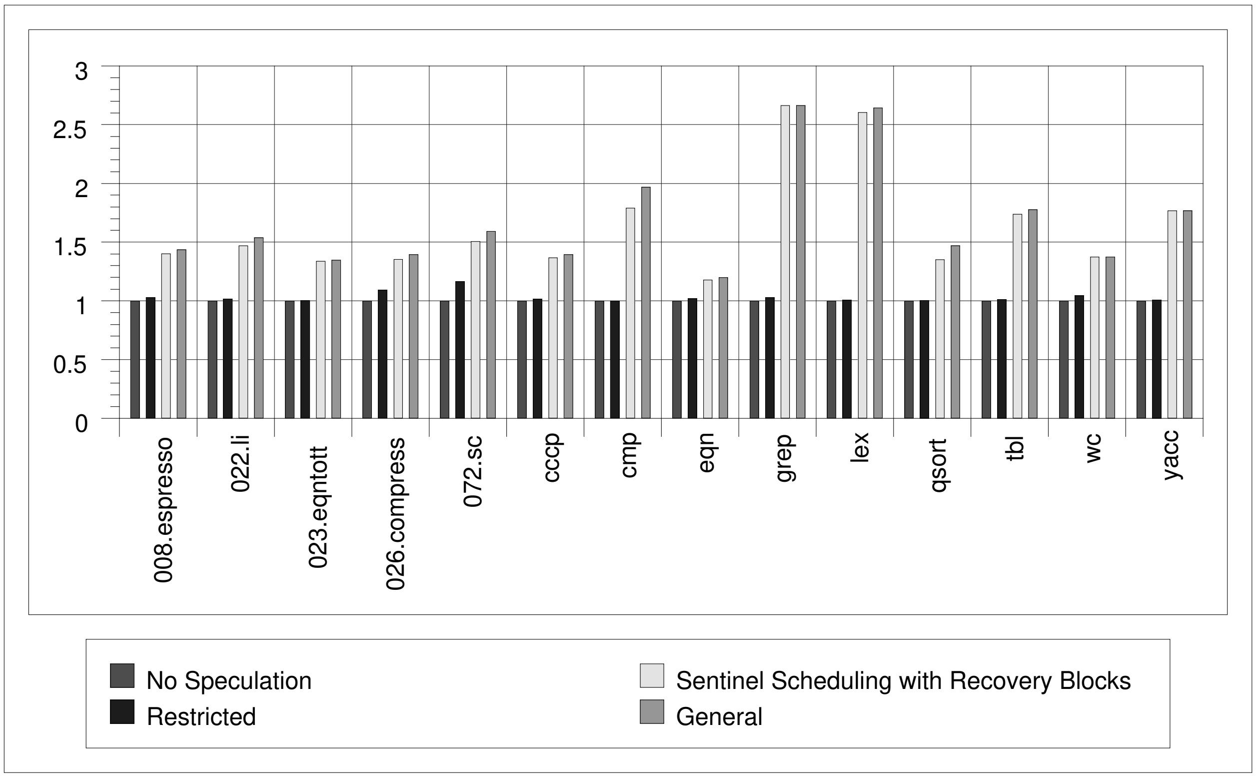

Figure 2: Relative performance of sentinel scheduling with recovery blocks.

It is clear that general speculation far outperforms restricted speculation. In fact, restricted only yields significant speedup in 026.compress and 072.sc. This is mostly due to the fact that no memory operations are speculated with the restricted model. Loads usually begin dependence chains, therefore, not being able to speculate them does not allow the overlap of more dependence chains. Dependence height becomes the limiting factor, reducing overall ILP.

General speculation would be ideal if it could detect exceptions. Sentinel scheduling with recovery blocks can detect and recover such exceptions, and it perform nearly as well as general. It performs 90.92% to 100.0% as well as general. On average, it yields a 97.14% speedup compared to general. At these levels, performance gain over restricted speculation is significant. Any difference between SSRB and general could due to additional explicit checks or spill code created by the added register pressure associated with speculation. The use of the destination register to hold a source register was significant in reducing the amount of spill code. It should also be noted that our implementation of SSRB used an oversimplified algorithm to choose which source register should be saved. It is our opinion that further gains could be made by improving this algorithm.

While Figure 2 does show good performance relative to general, it does not show the cumulative multiplicative effect of an aggressive optimizer combined with speculation. For example, the IMPACT compiler boosts the speed of cmp by roughly 5 times on the same machine without any speculation using only other optimizations. With SSRB, cmp gives 80% speedup over no speculation. The combined effect is that cmp will run roughly 900% of its original speed. Clearly, the positive effect of speculation and of SSRB is understated by the figure.

Code growth due to explicit checks can have a performance effect, however it more likely has a significant effect on the instruction cache. To demonstrate that our assumption of perfect instruction cache is valid, section 5.3 discusses the amount and effect of SSRB's code growth.

5.3 Code Growth

Sentinel scheduling with recovery blocks increases codes size in several ways. The largest and most obvious source of code growth is due to the recovery blocks themselves. SSRB also adds an explicit check when an implicit check cannot be found in the appropriate home block. In our model, each unprotected register generates one explicit check instruction. However, a real implementation of SSRB would allow several register operands for an explicit check, cutting down on the number of check instructions generated. In cases of speculated self-antidependencies, a renaming instruction must sometimes be created. Again, our implementation is naive. A refined version of the compiler could reduce code expansion due to added renaming instructions because not all of the ones generated are necessary.

In addition to the instructions added by our SSRB implementation, the process of speculation can increase register pressure. While register pressure is alleviated by the use of the destination register to allow reuse of one source register, an effect is still present. The register allocator copes with the added register pressure by creating the appropriate spill code. This spill code is another source of code growth.

Despite the naive aspects of certain parts of our implementation, the effect of any extra code growth is small. This is due to the fact that most of the code growth is contained in the recovery blocks themselves. The code in the recovery blocks is very rarely executed and therefore would

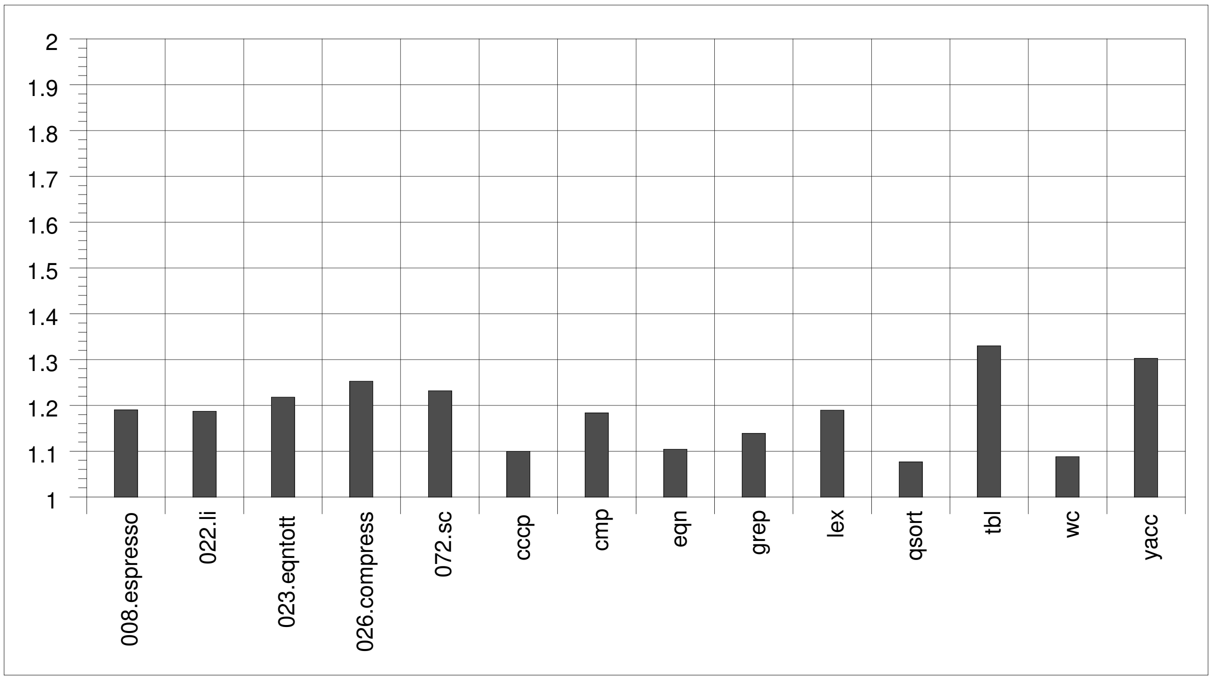

Figure 3: Size of code after sentinel scheduled with recovery blocks.

not normally cause instruction cache conflict misses with the normally executing code. A recovery block is only entered in the event of an exception. Most programs, including all the benchmarks here, terminate upon reaching an exception. Programs which would recover from an instruction's exceptions frequently could be compiled so frequently excepting instructions are not speculated. Work in this area is actively being pursued by the IMPACT compiler team. [16]

The actual code growth of each benchmark tested is presented in Figure 3. Code growth is normalized with respect to the code generated using the general speculation model. The code size grew anywhere from 8% to 33%. On average, 19% code growth was observed. The worst offenders were tbl and yacc which both had many frequently unrolled loops in which the scheduler performed aggressive speculation creating large recovery blocks and large amounts of spill code.

Code growth due to explicit checks is shown in Table 6. Table 6 shows the actual static code size and number of explicit checks for each benchmark. A more advanced implementation of SSRB should yield fewer explicit checks. Also, for an 8-issue processor, it is not be difficult to find open slots where these instructions can be scheduled reducing any effect they would have on execution

Benchmark	Assembly Instructions	Check Instructions
008.espresso	89191	1454
022.li	22653	234
023.eqntott	11555	149
026.compress	4525	105
072.sc	34722	666
cccp	13783	121
cmp	1025	31
eqn	11779	68
grep	1998	47
lex	24275	279
qsort	758	10
tbl	23368	580
wc	578	2
yacc	31070	697

Table 6: Maximum number of explicit sentinels required with 64 integer and 64 floating-point registers.

time.

6 Conclusion

The intent of this paper was to present and evaluate a better alternative to the currently available methods of safe speculation. In order for a method to be seriously considered for commercial implementation in a wide issue superscalar or VLIW processor, it must yield a large performance benefit and it must also be low cost. As was we have shown, Sentinel scheduling with recovery blocks clearly meets these requirements by having the lowest hardware cost of any method presented, while achieving good performance results.

Further work in sentinel scheduling with recovery block should be performed to tune its abilities even more. As was discussed earlier, more aggressive SSRB algorithms could be used in the scheduler to reduce code size and increase execution time.

Another area of work which could be explored deals with reducing any register pressure created. For any speculation model, a large portion of the speculated instructions are loads. One way to save register pressure with speculated loads could be to put the calculated memory address in the destination operand instead of only the first source operand. This works because loads except while attempting to load the already calculated addresses. Depending on the architecture, this could allow two or more registers to be reused, reducing register pressure and the spill code associated with it. While we did not implement this or study its effects, we feel that this should be examined in the future.

Finally, since sentinel scheduling with recovery blocks does not depend on a single speculation path, it would be worthwhile to evaluate the performance advantage this would yield when used with compilation methods other than superblock formation.

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] 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.

[4] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture*, pp. 183–198, October 1981.

[5] M. S. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 318–328, June 1988.

[6] A. Aiken and A. Nicolau, "Optimal loop parallelization," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 308–317, June 1988.

[7] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, vol. 22, pp. 12–35, January 1989.

[8] P. Y. Hsu and E. S. Davidson, "Highly concurrent scalar processing," in *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 386–395, June 1986.

[9] 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.

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

[11] 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*, pp. 248–259, October 1992.

[12] S. A. Mahlke, *Exploiting Instruction Level Parallelism in the Presence of Conditional Branches.* PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[13] R. A. Bringmann, S. A. Mahlke, R. E. Hank, J. C. Gyllenhaal, and W. W. Hwu, "Speculative execution exception recovery using write-back suppression," in *Proceedings of 26th Annual International Symposium on Microarchitecture*, December 1993.

[14] 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.

[15] Hewlett-Packard Company, Cupertino, CA, *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 1990.

[16] R. A. Bringmann, *Compiler-Controlled Speculation.* PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1995.