Speculative Execution Exception Recovery using Write-back Suppression

Roger A. Bringmann   Scott A. Mahlke   Richard E. Hank
John C. Gyllenhaal   Wen-mei W. Hwu

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

Correspondent: Roger A. Bringmann
Tel: (415)-336-4729
Email: roger@crhc.uiuc.edu

Abstract

Compiler-controlled speculative execution has been shown to be effective in increasing the available instruction level parallelism (ILP) found in non-numeric programs. An important problem with compiler-controlled speculative execution is to accurately report and handle exceptions caused by speculatively executed instructions. Previous solutions to this problem incur either excessive hardware overhead or extra register pressure. This paper introduces a new architecture scheme referred to as *write-back suppression*. This scheme systematically suppresses register file updates for subsequent speculative instructions after an exception condition is detected for a speculatively executed instruction. We show that with a modest amount of hardware, write-back suppression supports accurate reporting and handling of exceptions for compiler-controlled speculative execution without adding to the register pressure. Experiments based on a prototype compiler implementation and hardware simulation indicate that ensuring accurate handling of exceptions with write-back suppression incurs very little run-time performance overhead.

Index terms - exception detection, exception recovery, scheduling, speculative execution, VLIW, superscalar

1 Introduction

Compiler-controlled speculative execution is a technique that allows instructions to be scheduled above conditional branches that determine their execution in the original program order. When an instruction is speculatively scheduled above a conditional branch, it is executed regardless of the direction taken by the conditional branch. Therefore, speculative execution trades off execution efficiency for more scheduling opportunity across basic blocks. This technique is important since insufficient parallelism has been shown to exist in basic blocks of non-numeric benchmarks to keep the functional units busy for wide issue superscalar and VLIW architectures [4, 5, 6]. Global scheduling techniques such as *trace scheduling* [7] and *superblock scheduling* [8] take advantage of speculative execution to increase the available instruction level parallelism found in programs.

To ensure correct program execution, the compiler must not alter the program behavior 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 [8]. A more difficult requirement is that if the compiler speculates a potentially excepting instruction (*PEI*) from one path of a conditional branch, the resolution of its exception must not alter the program state if the branch chooses the other path. In particular, the exception conditions from speculatively executed instructions must not terminate the execution of the program unless their execution is confirmed by the subsequent branches.

The *general percolation model* [6] poses one solution to resolving exceptions for speculated potentially excepting instructions (*SPEI*s). This model adds a non-trapping instruction to the instruction set for each of the *PEI*s that it permits to speculate. Whenever a PEI is to be speculatively scheduled, it is changed into its non-trapping counterpart. During run time, the hardware ignores most of the exception conditions for the non-trapping instructions. However, page faults and TLB misses due to speculatively executed instructions will still have to be handled immediately, which can potentially increase the number of page faults and TLB misses during execution. This model is easy to implement and can accomplish high performance with minimal changes to the architecture. Unfortunately, it permits some exception conditions that would be detected in the original program execution to go undetected.

In this paper, a set of architectural features, collectively referred to as *Write-back Suppression (WBS)* is introduced. WBS provides an effective support for compiler-controlled speculative execution without excessive hardware overhead. We show that WBS incurs very little performance overhead while allowing accurate detection and reporting of exceptions that occur for speculatively executed instructions.

The next section provides background for compiler-controlled speculative execution. Section 3 introduces write-back suppression and the compiler algorithms to take advantage of it. Section 4 presents experimental results. Concluding remarks are given in Section 5.

2 Background

2.1 Problem Statement

Figure 1a illustrates the basic problems that compiler-controlled speculative execution schemes must address to accurately report and handle exceptions. In this example, 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 the memory location contained in r5 to perform a computation. 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. 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. However, since the branch is usually not taken, it is feasible to speculate I_1 above I_2 producing the schedule shown in Figure 1b (an instruction denoted by *(S)* is speculated). This new schedule has a 3 cycles latency and would be a more desirable schedule.

For the schedule in Figure 1b to be legal, it must not alter the program behavior regardless of

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 1: (a) un-scheduled code, (b) one speculated instruction, (c) two speculated instructions.

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 the exception until the direction of the branch is known. 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_1 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_1 caused the exception. Once we know that the branch is not taken, we must re-execute I_1 and permit the exception to occur. This allows the exception to be resolved naturally.

The problem becomes even more complicated if the scheduler decided to also move I_3 above I_1 as shown in Figure 1c. 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 handle exceptions for *SPEI*s: detecting a delayed exception, determining the instruction that caused the exception, protecting source operands until the exception is resolved and recovering from the exception.

2.2 Instruction Boosting

Instruction boosting has been proposed for handling exceptions with compiler-controlled speculative execution [1] [3]. 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 [3]. We will demonstrate in Section 3 that WBS solves the same problems without incurring either excessive hardware overhead or large code expansion.

2.3 Sentinel Scheduling

An alternative scheme to enable exception detection and recovery with compiler-controlled speculative execution is sentinel scheduling [2]. 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

I	Instruction		Current Block	Speculation Distance	Home 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 2: Original code segment.

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	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: Code segment after sentinel scheduling, before register allocation.

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	r10,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(r10),r4		2	0	2
12	mov	r8,r10		2	0	2

Figure 4: Code segment after sentinel scheduling and register allocation.

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 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 2. 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. Speculative instructions are denoted by (S) and the number of branches they moved above is specified by their speculative distance. 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 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.

[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

The code segment after register allocation is shown in Figure 4. In this example, a total of 11 physical registers are required to ensure exception detection and recovery are possible for all speculative instructions.

3 Write-back Suppression Scheduling

In this section, an architecture support for compiler-controlled speculative execution referred to as *write-back suppression* is introduced. Write-back suppression 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 the direction of the branch is known, and prevent corruption of the source operands of instructions by systematically suppressing updates to the register file after an exception from a SPEI.

3.1 Overview

A few key terms will be introduced using the examples in Figures 2 and 5. The *home block* of an instruction is the basic block where the instruction was located prior to scheduling. Thus, 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 block the instruction is located in after scheduling. As Figure 5 shows, I_6 has been speculated into current block 0. The *Speculation distance* of an instruction is the number of branches that an instruction was speculated beyond during scheduling. As the figure shows, I_6 has a speculation distance of 1.

The instruction schedule shown in Figure 5 contains only minor modifications from the schedule in Figure 3 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

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 5: Code segment after WBS scheduling, before register allocation.

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 begin recovery.

As discussed in Section 2.1, in order for the delayed excepting instruction to accurately recover to the correct processor state, the source operands of speculated instructions must be protected during the exception. 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 5, 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 were to both except when no other exceptions are pending, we will have *nested exceptions.* This occurs because I_9 (from home block 2) can only suppress I_{10} (from home block 2)

I	Instruction			Current Block	Speculation Distance	Home 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 6: Code segment after WBS scheduling and register allocation.

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}. This 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 at this point. 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}.

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 6, shows, I_6 is able to overwrite one of its own source operands and still recovery 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 6 using only 7 registers while sentinel scheduling requires 11 registers. As this example shows WBS can reduce the register pressure which will result in less spill code.

3.2 Compiler Support

This section outlines the changes made to the scheduler and the register allocator to support WBS.

3.2.1 Scheduler Extensions

There are two major steps for scheduling in IMPACT - the dependence graph construction and list scheduling. After the dependence graph is built and reduced [2], 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 belong to the same home block of 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 3.3. As described in Section 3.3, a PEI from K home blocks later will report its exception to the same recovery array location as a PEI from the home block of the check instruction. This would cause an erroneous recovery condition. These dependences ensure that after scheduling any of these PEIs will be located after the check instruction.

The list scheduler has been modified to add additional information to speculated PEIs to permit correct register allocation. The live ranges of the source operands of SPEIs are extended down to the last instruction that is from a home block earlier than the home block of the SPEI (the benefits of this will be discussed in the Section 3.2.2).

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 checks input dependencies 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 5 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.

3.2.2 Register Allocator Extensions

Register allocation in our compiler is done using a global graph coloring approach [9]. 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 in two ways to ensure that the resulting allocation will allow proper recovery in the event of an exception. These modifications prevent the register allocator from destroying the source operands of an SPEI. This 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.

This first modification involves live range construction. 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 each SPEI with the last instruction from a home block that is above the SPEI's home block. The register allocator adds instructions to the live range that lies 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 5. 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 the 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 6 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.

The second modification involves the handling of operand spilling. When the register allocator is unable to allocate an operand of an SPEI, to a register, the register allocator un-speculates the SPEI to reduce the register pressure. The speculated instructions responsible for the increase in register pressure are moved downward one block at a time until one of two events occurs. If the live range of the destination of the speculated instruction becomes allocatable, the downward code motion ceases. The downward code motion also stops once the speculated instruction has been moved below the last instruction from a home block above that of the SPEI indicated by the scheduler. At this point if the destination operand is not allocatable, the register allocator spills the operand without affecting recovery. By incrementally un-speculating the instructions responsible for the increased register pressure, the register allocator allows as much speculation as is possible for a given number of available registers while introducing as little spill code as possible.

3.3 Architectural Extensions

In order to support write-back suppression scheduling, several extensions are required to the architecture. These extensions are broken down into three groups, instruction set extensions, extensions to support suppression of register file updates, and extensions to support recovery from delayed exceptions. Each of these extensions will be discussed in light of how they support the requirements of WBS as described in Section 3.1.

Each instruction opcode will be augmented with a *k-bit* field speculation distance 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 value for k was chosen after analyzing a series of speculatively scheduled benchmarks. This will be outlined in detail in Section 4.2.1.

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.

When a branch instruction retires, it increments the *current_block* register. If the branch is taken, it causes the suppression hardware and recovery hardware to reset to prevent an exception from propagating into another superblock.

Figure 7 depicts the hardware extensions required by a processor to suppress register file updates during the retire stage of an out-of-order completion architecture. The suppression hardware is inserted between the reorder buffer and the register file to control the *write-enable* flag of the register file. The suppression circuit shown is a k-bit comparator with some additional combinatorial logic. This circuit is responsible for determining if the instruction currently being retired is allowed to update the register file.

A CHECK flag and a SPECULATED flag have been added to the *flags* field of each reorder buffer entry. These flags along with the EXCEPTION flag make the suppression circuit switch between the *normal state*, *suppression state*, and the *recovery state*. The normal state allows all instructions to update the register file. The suppression state is entered from the normal state when the SPECULATED and EXCEPTION flags are set for the instruction currently being retired. The recovery state is entered when the CHECK flag is set for an instruction and there is a delayed exception existing for its home block. These states are maintained in the block *state* at the top of the figure.

The *adder* above the reorder buffer is used to compute the home block of the retiring instruction every cycle. It takes the speculated distance field from the reorder buffer and adds it to the

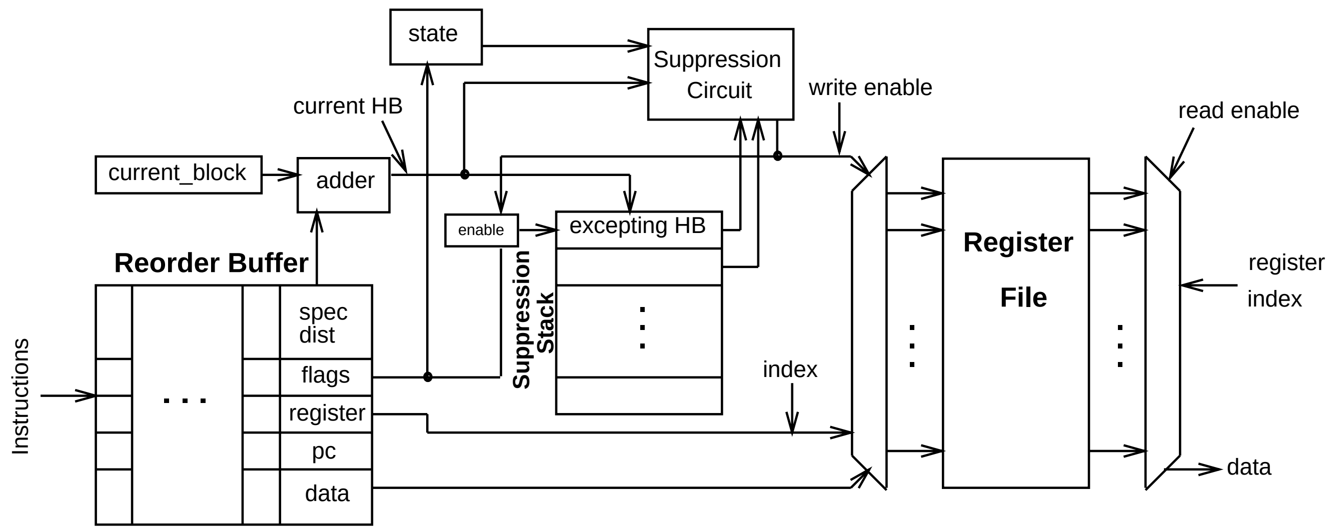

Figure 7: Write-back suppression hardware.

current_block register.

Starting in the normal state, an instruction whose SPECULATED and EXCEPTION flags are set will put the suppression hardware into the suppression state, push the computed home block onto the top of the *suppression stack* and place its PC into the *recovery array* at the entry indexed by its home block (Figure 8). Its update to the register file will then be suppressed.

While in the suppression state, the computed home block of each retiring instruction and the excepting home block 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 entry 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 3.1, a nested exception occurs when an excepting SPEI retires with a computed home block earlier than the current excepting home block. This is maintained by pushing the new higher priority exception onto the top of the suppression stack.

A check instruction is permitted to initiate recovery if its respective entry in the recovery array is valid. This allows WBS to determine that there is a delayed exception. Figure 8 shows the

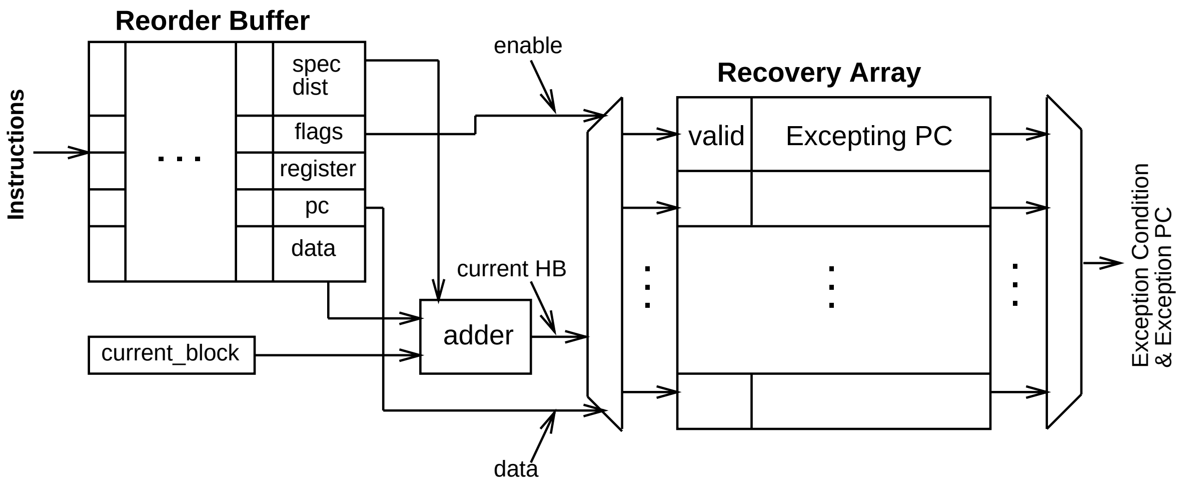

Figure 8: Delayed exception recovery hardware.

hardware designed to permit recovery from the delayed exception. The only way for an exception to be recorded is when the speculated excepting instruction retires. Since the instruction has been released from the reorder buffer, the *recovery array* is used to keep track of the address of the excepting instruction until its check is encountered. The home block of the check instruction is used to index into the recovery array to determine the address to begin recovery. The excepting PC from the array is provided to the fetch stage of the processor. When the this excepting instruction retires the second time, it is permitted to except.

After the excepting instruction resolves its exception, suppression hardware transitions into the recovery state. During this state, all retiring instructions whose computed home block is less than the excepting home block on the top of the suppression stack will be inhibited from updating the register file. Any retiring instructions whose computed home block is greater than or equal to the home block of the excepting home block in the second entry of the suppression stack will also be inhibited from updating the register file. All other instructions will be permitted to update the register file. The use of excepting home blocks found in the top two entries of the suppression stack ensures the correct nested recovery requirements described earlier. Once recovery is complete, the top-most entry of the suppression stack will be popped. If the suppression stack is empty, the suppression state will transition to the normal state, otherwise it will transition back to the suppression state.

4 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 general percolation and sentinel scheduling.

4.1 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 [6]. The benchmarks used in this study are the 14 non-numeric programs shown in Table 1. The benchmarks consist of 5 non-numeric programs from the SPECint92 suite and 9 other commonly used non-numeric programs.

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 1: Benchmarks

The processor model used in this study is a in-order issue superscalar processor with register interlocking. The processor is assumed to have uniform function 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 2). The processor is assumed to trap on exceptions for memory load, memory store, and all floating point instructions.

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 2: Instruction latencies.

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.

4.2 Results

4.2.1 Selection of the Maximum Speculation Distance

The general percolation 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, 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 9. Considering that a speculation limit of 2^k-1 branches is possible given the addition of k bits to 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.

4.2.2 Comparison of WBS and General Percolation

The performance results of the WBS scheduling model and general percolation scheduling are shown in Figures 10 through 13. All numbers are shown as a percentage of the the performance, defined as one over cycle count, of general percolation. The figures show a general improvement

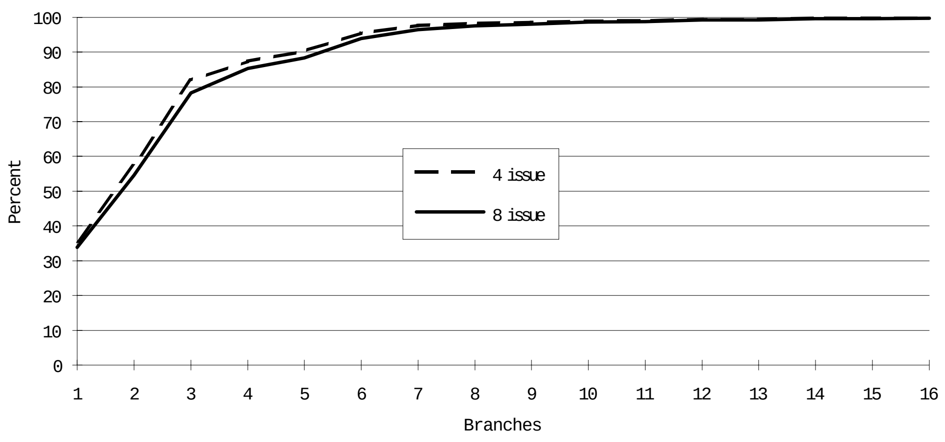

Figure 9: Weighted speculation distances for *PEI*s.

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. One exception to the trend is shown in Figure 13 by grep which is the most parallel benchmark in the experiments. The additional check instructions 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. One notable exception is compress shown in Figure 12 which shows a decrease in the performance from 32 to 64 registers. The schedule generated for compress using general percolation suffers from a great deal of register pressure with only 32 integer registers. However, the register allocator will not spill any live ranges for SPEI using WBS since this would open the potential for an unrecoverable exception. Instead, the register allocator demotes the SPEI until the register pressure is eliminated. The spill code inserted for the compress using general percolation increases the cycle count and therefore penalizes compress which artificially improves the relative performance for WBS.[2] As a side effect, compress with 64 registers with WBS appear to perform worse.

[2]The register allocator will be modified for the final draft of the paper to demote a live range of a speculated excepting instruction instead of spilling them in order to remedy this problem.

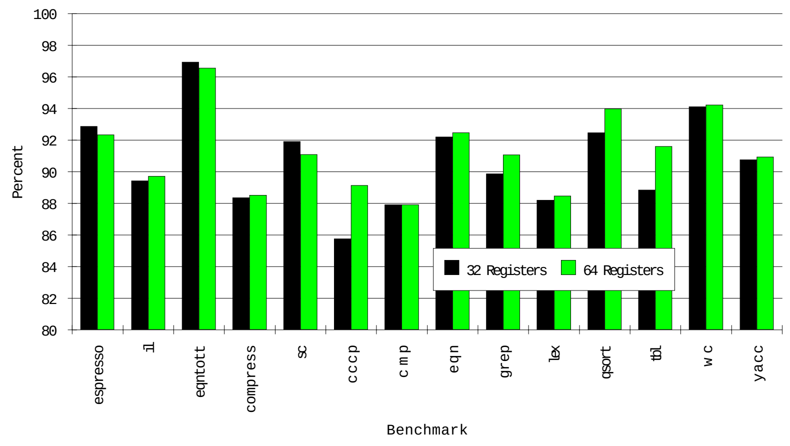

Figure 10: Performance results of WBS relative to general percolation using an issue 1 processor.

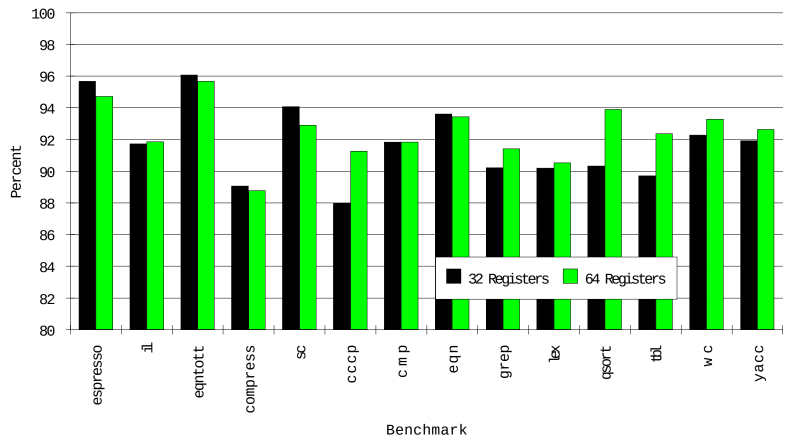

Figure 11: Performance results of WBS relative to general percolation using an issue 2 processor.

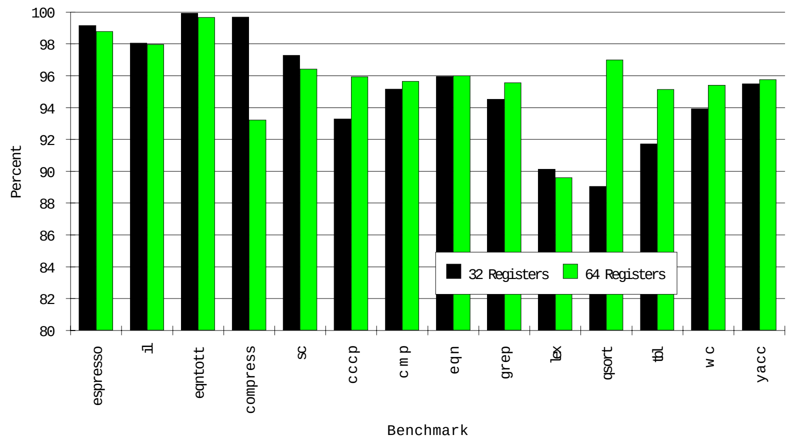

Figure 12: Performance results of WBS relative to general percolation using an issue 4 processor.

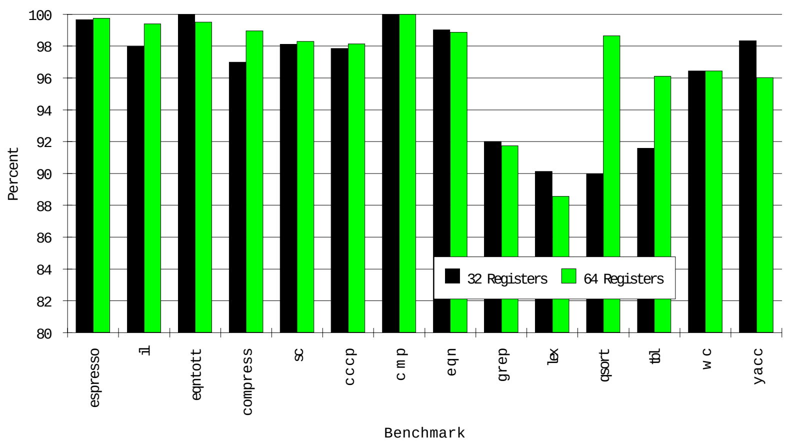

Figure 13: Performance results of WBS relative to general percolation using an issue 8 processor.

4.2.3 Comparison of WBS and Sentinel Scheduling

These numbers were unavailable at the time of submission. They will be included in the final draft of the paper.

5 Conclusion

This paper has introduced a new architecture scheme referred to as *write-back suppression (WBS)*. 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 without adding to the 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 88 to 100 percent of the performance gains of general percolation scheduling and still ensure correct execution under all conditions.

Future research will study the benefits of WBS scheduling with existing instruction set architectures by adding a few new instructions to the instruction set instead of extending each opcode to include the speculation distance. The benefits of WBS scheduling will also be studied for architectures with non-uniform functional units. Finally, the cache effects of WBS scheduling will be studied using recovery blocks and the in-line recovery method discussed in this paper.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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 structure for VLIW and superscalar compilation," tech. rep., Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, February 1992.

[9] 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.