Sentinel Scheduling for VLIW and Superscalar Processors

Scott A. Mahlke William Y. Chen Wen-mei W. Hwu

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

B. Ramakrishna Rau Michael S. Schlansker

Hewlett Packard Laboratories
Palo Alto, CA 94303

Abstract

Speculative execution is an important source of parallelism for VLIW and superscalar processors. A serious challenge with compiler-controlled speculative execution is to accurately detect and report all program execution errors at the time of occurrence. In this paper, a set of architectural features and compile-time scheduling support referred to as *sentinel scheduling* is introduced. Sentinel scheduling provides an effective framework for compiler-controlled speculative execution that accurately detects and reports all exceptions. Sentinel scheduling also supports speculative execution of store instructions by providing a store buffer which allows probationary entries. Experimental results show that sentinel scheduling is highly effective for a wide range of VLIW and superscalar processors.

1 Introduction

Instruction level parallelism (ILP) within basic blocks is extremely limited. An effective VLIW or superscalar machine must schedule instructions across basic block boundaries to achieve higher performance. When branch conditions may be determined early, scheduling techniques such as software pipelining [1] [2] [3] are effective for exposing ILP. Also, predicated instructions can be used in conjunction with software pipeline loop scheduling [4] or straight-line code scheduling [5] to mask out the effects of unnecessary instructions from alternate paths of control. For applications in which branch conditions cannot be determined early, speculative execution of instructions is an important source of ILP [6] [7] [8].

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*. Speculative execution may either be engineered at run-time using dynamic scheduling or at compile-time. This paper focuses on compile-time engineered speculative execution, or speculative code motion.

There are two problems associated with speculative code motion. The first problem is that the result value of a speculative instruction that was not required to execute must not affect the execution of the subsequent instructions. This can be effectively achieved by compile-time renaming transformations. A more serious problem with speculative execution is correctly detecting exceptions that occur. An exception that occurred for a speculative instruction which was not supposed to execute must be ignored. On the other hand, an exception for a speculative instruction that was supposed to execute must be signaled. Accurately detecting and reporting exceptions are required to identify program execution errors at the time of occurrence. Also, recovery from an excepting speculative instruction should be possible.

In this paper, a set of architectural features and compile-time scheduling support, collectively referred to as *sentinel scheduling*, is described. Sentinel scheduling provides an effective framework for speculative execution, while also providing a means to accurately detect and report exceptions that occur for speculatively executed instructions.

2 Background and Related Work

Varying degrees of speculative code motion can be supported with different scheduling models. In this section, three existing scheduling models, restricted percolation, instruction boosting, and general percolation, along with their support for detecting and reporting exceptions are discussed. An efficient structure to perform scheduling across basic blocks is a superblock. All scheduling techniques in this paper will be described based on the superblock structure, however they can be easily generalized to other structures. For example, trace scheduling [9], modulo scheduling [1], and enhanced pipelining [10] may effectively utilize the speculative execution models discussed in this paper. Tirumalai *et al.* showed that modulo scheduling of while loops depend on speculative support to achieve high performance [7]. Without speculative support, dependences limit the amount of execution overlap between loop iterations.

2.1 Superblock Scheduling

Superblock scheduling is an extension of trace scheduling [9] which reduces some of the bookkeeping complexity [8]. A superblock is a block of instructions in which control may

0

only enter from the top but may leave at one or more exit points. Superblock scheduling consists of two steps, dependence graph construction and list scheduling. The dependence graph represents the control and data dependences between instructions within a superblock. Control dependences are used to reflect to major restrictions to speculatively moving or percolating an instruction, I, before a branch, BR: (1) the destination of I is not used before it is redefined when BR is taken,[1] and (2) I will not cause an exception that alters the execution result of the program when BR is taken.

The different code scheduling models observe varying combinations of the two restrictions. For all scheduling models, restriction (1) can be overcome by compile-time renaming transformations. After the appropriate control dependences are eliminated according to the model used, list scheduling using the dependence graph, instruction latencies, and resource constraints is performed to determine which instructions are scheduled together.

2.2 Restricted Percolation Scheduling Model

The scheduler enforces both restrictions (1) and (2) when using the restricted percolation scheduling model [8]. Thus, only instructions which the compiler can guarantee to never cause execution-altering exceptions are candidates for speculative code motion. For conventional processors, memory load, memory store, integer divide, and all floating point instructions are potential trap-causing instructions. With these constraints, conventional exception detection does not need to be altered with this scheduling model. The limiting factor of restricted percolation is the inability to move potential trap-causing instructions with long latency, such as load instructions, above branches.

2.3 Instruction Boosting Scheduling Model

The scheduler enforces neither restriction when using the instruction boosting scheduling model [6]. The restrictions are overcome by providing sufficient hardware storage to buffer results until the branches an instruction moved past are committed. If all branches are found to be correctly predicted, the machine state is updated by the boosted instructions' effects. If one or more of the branches are incorrectly predicted, the buffered results are thrown away. Two sets of buffer storage are required for this scheduling model, shadow register files and shadow store buffers. The shadow register files hold the results of all boosted instructions which write into a register, while the shadow store buffers hold the results of all boosted store instructions. To boost an instruction above N branches, N shadow register files and N shadow store buffers are required. Therefore, the number of branches an instruction can be boosted above is limited to a small number.

Exceptions for boosted instructions are detected by marking in the appropriate shadow structure whether an exception occurred during execution. Then, when the machine state is updated for a correctly predicted branch, exceptions that occurred are signaled. An exception that occurred for a boosted instruction whose result is thrown away is ignored.

2.4 Ignoring Exceptions with the General Percolation Scheduling Model

The scheduler removes restriction (2) using the general percolation model [8]. Exceptions that may alter program execution are avoided by converting all speculative instructions which potentially cause traps into non-trapping or silent versions of those instructions. Memory stores, though, are not allowed to be speculative instructions. In order to support this scheduling model, an instruction set must contain a silent version of all trapping opcodes. When an exception occurs to a silent instruction, the memory system or function unit simply ignores the exception and writes a garbage value into the destination register. The consequence of using this value is unpredictable, and is likely to lead to a later exception or an incorrect execution result.

The inability to always detect exceptions and determine the excepting instruction limits the application of this scheduling model. Colwell *et al.* detect some exceptions by writing NaN into the destination register of any non-trapping instruction which produces an exception [11]. The use of NaN is then signaled by any trapping instruction. This method, however, has difficulties determining the original excepting instruction, and is not guaranteed to signal an exception if the result of a speculative exception-causing instruction is conditionally used. Also, an equivalent integer NaN must be provided for this method to work for integer instructions.

In summary, instruction boosting provides an effective framework for speculative code motion of instructions and identification of exceptions that occur for speculative instructions. However, the hardware overhead is very large, and the number of branches an instruction can be boosted above is limited to a small number. General percolation, on the other hand, achieves nearly the same performance level of instruction boosting [8] with a much lower implementation cost. The problem is that there is no guarantee of detecting exceptions and determining the cause of an exception. In the next section, a new scheduling model referred to as *sentinel scheduling* is introduced. With a modest amount of architectural support, sentinel scheduling permits all the scheduling freedom of general percolation, while allowing exceptions to be always detected and the excepting instruction accurately identified.

3 The Sentinel Scheduling Model

In this section, a scheduling model referred to as *sentinel scheduling* is introduced. Sentinel scheduling combines a set of architectural features with sufficient compile time support to accurately detect and report exceptions for compiler-scheduled speculative instructions. The basic idea behind this technique is to provide a *sentinel* for each potential exception-causing instruction that is speculatively executed. The sentinel can either be an existing instruction in the program or a newly created instruction. The sentinel reports

[1] Note that instructions in a superblock are placed sequentially by the compiler, therefore instructions following a conditional branch within a superblock are in the branch's fall-through path.

any exceptions that were caused by the speculative instruction. In the following subsections, the model of execution, the required architectural support, the algorithm for sentinel scheduling, and several important issues are described.

3.1 Model of Execution

Conceptually, each instruction, I, can be divided into two parts, the non-excepting part that performs the actual operation, and the sentinel part that flags an exception if necessary. The non-excepting part of I can be speculatively executed, provided the sentinel part of I remains in I's original basic block or *home block*. The sentinel part of I can be eliminated if there is another instruction, J, in I's home block which uses the result of I. The sentinel part of J will signal any exceptions caused by both I and J, which makes it a shared sentinel between I and J. This argument can be applied recursively until an instruction that has no use in its home block is encountered.[2] Such an instruction is termed an *unprotected instruction*. If an unprotected instruction is speculatively executed, an explicit instruction must be created to act as the sentinel part of that instruction. The explicit sentinel is restricted to remain in the instruction's home block.

Since some instructions may never result in exceptions, e.g., integer add, the sentinel part of some unprotected instructions is not necessary. The sentinel part of an unprotected instruction which cannot cause an exception is only necessary if it is used to report an exception for a previous speculative instruction.

3.2 Architectural Support

In order to support sentinel scheduling, several extensions are required to the processor architecture. The first extension is an additional bit in the opcode field of an instruction to represent a speculatively executed instruction. This additional bit is referred to as the speculative modifier of the instruction. The compiler sets the speculative modifier for all instructions that are moved above one or more branches. A second extension is an exception tag added to each register in the register file. The exception tag is used to signal an exception that occurred when a speculative instruction is executed.[3] The exception tag associated with each register must be preserved along with the data portion of that register whenever the contents of the register are temporarily stored to memory, e.g., register spill, function call, or context switch. The third extension is to provide special load and store instructions to save both the exception tag and data of a register. These instructions do not signal exceptions according to the exception tag in order to facilitate saving/restoring registers containing an exception condition.

[2] Note that a post dominating use is sufficient to guarantee all exceptions will be detected. However, a use in the home block is required in our implementation to facilitate earlier reporting of exceptions, re-executing less instructions for recovery, and reducing register lifetimes.

[3] Note that the minimum exception tag required is a single bit. However, in many cases a larger tag is useful to indicate the type of exception to assist in debugging and exception handling.

A summary of exception detection using the sentinel scheduling model is shown in Table 1. For each instruction, I, three inputs are examined, the speculative modifier of I, the exception tag of the source registers of I, and whether I results in an exception. A single bit is used for the exception tag to simplify this discussion.

Execution of a Speculative Instruction. When I is a speculative instruction, exceptions will not be signaled immediately. If all the source register exception tags of I are reset, conventional execution results when I does not cause an exception. When I does cause an exception, the exception tag of the destination register is set, and the program counter (pc) of I is copied into the data field of the destination register. The pc of I can be obtained from a PC History Queue which keeps a record of the last m pc values to enable reporting exceptions with non-uniform latency function units [11] [4]. If one or more of the source register exception tags of I are set, an exception propagation occurs. This is independent of whether I causes an exception or not. For this case, the destination register exception tag is set and the data of the source register with exception tag set is copied into the destination register. If more than one of the source registers of I have their exception tag set, the data field of the first such source is copied into the destination register. The implications regarding this issue will be discussed in Section 3.6.

Execution of a Non-speculative Instruction. If I is not a speculative instruction, conventional execution results if all source registers have their exception tags reset. When I causes an exception, the exception is signaled immediately, and I is reported as the exception-causing instruction. Conversely, when one or more of the source register exception tags are set, an exception has occurred for a speculatively executed instruction for which I serves as the sentinel. The exception is, therefore, signaled and the data contents of the source register with its exception tag set is reported as the pc of the exception-causing instruction. Again, if more than one source register has its exception tag set, the data field of the first such source operand is reported as the pc of the exception causing instruction.

Additional Sentinel Instruction. The final extension to the processor is an additional instruction called $check\_exception(reg)$. This instruction is inserted as the explicit sentinel for unprotected instructions which are speculatively executed. This instruction does not perform any computation, but rather is merely used to check the exception tag of its source register. For most processors, a new opcode does not need to be created, but rather a move instruction can be used instead. The destination register of the move is either set to the same as the source register or to a register hardwired to 0, such as R0 in the MIPS R2000.

3.3 Sentinel Superblock Scheduling Algorithm

As previously discussed, code scheduling within a superblock consists of two major steps, dependence graph construction and list scheduling. An algorithm to perform sentinel superblock scheduling is included in the Appendix.

The first step of the scheduling algorithm is dependence graph construction and reduction. The initial dependence

spec	*src(I).except_tag* †	*I causes except.*	*dest(I).except_tag*	*dest(I).data*	*except. signal*
0	0	0	0	result of I	none
0	0	1	0	-	yes, except. pc = pc of I
0	1	0	0	-	yes, except. pc = $src(I).data$ ‡
0	1	1	0	-	yes, except. pc = $src(I).data$ ‡
1	0	0	0	result of I	none
1	0	1	1	pc of I	none
1	1	0	1	$src(I).data$ ‡	none
1	1	1	1	$src(I).data$ ‡	none

† union of all source operand exception tags of I ‡ the first source operand of I whose exception tag is set

Table 1: Exception detection with sentinel scheduling.

graph contains dependence arcs to represent all data and control dependences between instructions in the superblock. Dependence graph reduction removes control dependences between branches and instructions to enable speculative code motion allowed by the scheduling model. With the sentinel scheduling model, only restriction (1) (Section 2.1) is enforced. Therefore, a control dependence arc from a branch instruction, BR, to another instruction, I, is removed if the location written to by I is not used before being redefined when BR is taken. As with general percolation, memory stores are not allowed to be moved above branches. However in the next section, an extension to remove this constraint will be discussed. Dependence graph reduction also identifies those instructions in the superblock which are unprotected.

The second step of the scheduling algorithm consists of a modified version of list scheduling. The scheduler is modified so that when an unprotected instruction, I, is moved above a branch, a sentinel instruction is inserted into I's home block. Control dependences are added to restrict the sentinel to remain in I's home block. The sentinel is then added to the list of unscheduled instructions and list scheduling resumes. Finally, the compiler sets the speculative modifier of all those instructions that moved above a branch in the superblock.

3.4 Sentinel Scheduling Example

To illustrate sentinel scheduling and exception detection with sentinel scheduling, consider the assembly code fragment shown in Figure 1(a). For simplicity, it will be assumed in this example that each instruction requires one cycle to execute, and the processor has no limitations on the number of instructions that can be issued in the same cycle. Also, it will be assumed that memory loads and stores are the only instructions that may cause exceptions. After dependence graph reduction, instructions E and F are identified as unprotected, since they are the last uses of the potential trap-causing instructions, B and C, respectively.

The code segment after scheduling is shown in Figure 1(b). Four instructions (B, C, D, and E) are moved above a branch, therefore their speculative modifiers are set. Instruction E, though, is unprotected, so an explicit sentinel (instruction G) must be inserted into E's home block to serve as a sentinel. Since instruction F is not speculatively executed, an explicit sentinel is not inserted for it. In the final schedule, instructions F and G serve as sentinels for

```
  A: if (r2==0) goto L1
  B: r1 = mem(r2+0)
  C: r3 = mem(r4+0)
  D: r4 = r1+1
† E: r5 = r3×9
† F: mem(r2+4) = r4

† unprotected instruction
‡ sentinel
```

(a)

```
*   B[1]: r1 = mem(r2+0)
*   C[1]: r3 = mem(r4+0)
*   D[2]: r4 = r1+1
*   E[2]: r5 = r3×9
    A[3]: if (r2==0) goto L1
‡   F[3]: mem(r2+0) = r4
‡   G[3]: check_exception(r5)

*   speculative instruction
[n] indicates in which cycle
    the instruction is executed
```

(b)

Figure 1: Example of sentinel scheduling. (a) Original program segment. (b) Program segment after scheduling.

instructions B, C, D, and E.

An execution sequence for the scheduled code segment in which instruction B causes an exception is shown in Figure 2. For this example, it is assumed that the branch, instruction A, is not taken. The initial states of all the registers are further assumed to all have reset exception tags and some unknown data fields. In the first cycle, instruction B causes an exception, however since it is a speculative instruction, the exception is not yet signaled. Instead, the exception tag of the destination register of instruction B is set, and the pc of instruction B is copied into the destination register's data field. In the second cycle, instruction D finds the exception tag of its first source register set, however since it is also a speculative instruction, it propagates the exception information to its destination register. Finally, in cycle 3 instruction F detects that the exception tag of its first source register is set. Since instruction F is not a speculative instruction, an exception is signaled and the cause of the exception is reported as the contents of $r4$.

Note that in this example, if instruction B again results in an exception but the branch instruction A is instead taken, the exception is completely ignored. This result is correct because if the branch is taken, instruction B should not have been executed, and therefore should not disrupt the program's execution.

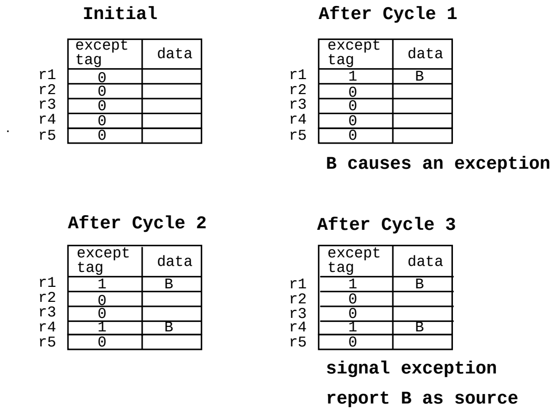

Figure 2: Example of exception detection using sentinel scheduling.

3.5 Handling Uninitialized Data

The use of an uninitialized register can potentially cause incorrect exceptions to be reported with the sentinel scheduling model. Registers which are not defined may have their exception tag set. The use of this register will therefore lead to an immediate or eventual exception signal. However, this exception should not be reported. To prevent an exception from occurring with uninitialized registers, the compiler performs live variable analysis [12], and inserts additional instructions to reset the exception tags of the corresponding registers before they are used.

3.6 Reporting Multiple Exceptions

Multiple exceptions in a program are handled efficiently with sentinel scheduling. The exceptions can either occur within different basic blocks or within the same basic block. When two exceptions occur in different basic blocks, the exceptions are guaranteed to be detected in the proper order because exceptions for all instructions of a basic block are checked before the basic block is exited. The requirement of a sentinel in the home block of each speculative instruction enforces this condition.

For multiple exceptions in the same basic block, exceptions are not guaranteed to be detected in the proper order according to the original code sequence. Multiple excepting instructions in the same basic block may either have different sentinels or share a sentinel. With different sentinels, the first sentinel executed will signal the first exception. When two excepting instructions share a sentinel, multiple source registers of the sentinel instruction will have their exception tags set. In this case, one of the exceptions is arbitrarily first signaled. If a recovery mechanism is utilized, as discussed in the next section, the second exception is reported when the sentinel is re-executed. The order of reporting two exceptions in the same basic block is difficult to maintain in many systems. In many cases, instructions within a basic block are reordered by conventional compiler code optimizations. Therefore, an order of reporting exceptions in the same basic block is not maintained with the sentinel scheduling model.

3.7 Recovery Issues

For some types of exceptions, it is desirable to retry the excepting instruction rather than abort program execution. With speculative instructions, this becomes more difficult because an excepting speculative instruction may have a chain of dependent speculative instructions which also require retrying. A comprehensive treatment of the appropriate policy, hardware support, and compiler conventions needed to permit recovery and retry are beyond the scope of this paper. The following technique is informally described in order to demonstrate that exception retry is possible with sentinel scheduling. A more complete treatment of this technique is presented in [13].

A restartable instruction sequence is a list of consecutive instructions that satisfy the following two constraints. First, none of the instructions in the sequence may cause irreversible side effects that prevent re-execution of any instructions in the sequence. As a result, I/O, subroutine call, and synchronization instructions break restartable sequences. These instructions will be referred to as *irreversible instructions.* In this paper, we assume a weak ordering memory architecture model where memory stores are not irreversible instructions. The second constraint is that the input operands to an instruction in a restartable instruction sequence are never overwritten by itself or by a subsequent instruction in the same sequence.

Scheduler Support. In order to be able to recover from all exceptions caused by speculative instructions, all instructions between a speculative instruction and the instruction which serves as its sentinel must form a restartable instruction sequence. This implies the following additional restrictions to the superblock sentinel scheduling algorithm.

1. A speculative instruction cannot be moved beyond any irreversible instruction. This is enforced by creating control dependence arcs from irreversible instructions to all subsequent instructions in the superblock.

2. Each irreversible instruction defines a basic block boundary as far as the sentinel scheduling algorithm is concerned. Therefore, all the speculative instructions that precede an irreversible instruction in the original program order will have a sentinel present before the irreversible instruction.

3. A speculative instruction cannot be moved beyond an instruction that modifies its own input register or memory location. This constraint, however, can be alleviated by the renaming transformations to be described below.

4. If an instruction I overwrites the input operands of one or more instructions that are scheduled after a speculative instruction, then I must be scheduled after the sentinel of the speculative instruction. Note that there may be multiple such speculative instructions whose sentinel must precede an overwriting instruction. Also, this restriction has to be enforced for both register and memory operands.

The code scheduler can perform renaming transformations to overcome restrictions 3 and 4 for speculative code motion. A typical application is to convert an increment of a register

```
  A: jsr
  B: r5 = mem(r3+0)
  C: if (r5==0) goto L1
† D: r1 = mem(r6+0)
  E: r2 = r2+1
  F: mem(r4+0) = r7
  G: r8 = r1+1
  H: r9 = mem(r2+0)
```

```
  A[1]:  jsr
† D[2]:  r1 = mem(r6+0)
  B[2]:  r5 = mem(r3+0)
  E'[2]: r10 = r2+1
  C[3]:  if (r5==0) goto L1
‡ G[4]:  r8 = r1+1
  F[5]:  mem(r4+0) = r7
  H'[5]: r9 = mem(r10+0)
  I[5]:  r2 = r10
```

† instruction considered for speculative execution
‡ sentinel for D
[n] indicates in which cycle the instruction is executed, each instruction requires 1 cycle to execute

(a) (b)

Figure 3: Example of sentinel scheduling to allow recovery. (a) Original program segment. (b) Program segment after scheduling and transformation.

into two instructions, one addition that writes into a new register, and one move instruction that updates the original register.[4] The move instruction is scheduled after the sentinel of the speculative instructions that are moved beyond the original increment instruction. All the uses of the original register within the superblock are renamed to the new register. With this transformation, the scheduler can always overcome restriction 3. Similarly, one can rename the destination register of an overwriting instruction specified in restriction 4 to enable a better code schedule.

Register Allocator Support. It is necessary to extend the live range of source registers for instructions subsequent to a speculative instruction to reach the sentinel for that speculative instruction. This ensures that the register allocator does not reuse these source registers and violate the restartable property enforced by the code scheduler. This technique assumes that speculative code motion is performed before register allocation. Also, it will tend to increase the number of registers used by the register allocator.

The scheduler and register allocator support can be illustrated with the code segment in Figure 3. Note that A is an irreversible instruction that blocks the movement of D due to restriction 1. In this example, we assume that the store instruction F may overwrite the input location of load instruction B. As a result, instruction F must be scheduled after G, the sentinel for speculative instruction D. Furthermore, to overcome restriction 3, the scheduler applies a renaming transformation on instruction E, which becomes E' and I in the final schedule. This transformation allows speculative instruction D to move beyond E'. As far as the register allocator is concerned, virtual register r10 must not be assigned to the same physical register as r2. This is achieved by extending the live range of r2 to G, the sentinel for instruction D.

To summarize, by enforcing constraints for speculative code motion and register allocation, one can guarantee that all instructions between a speculative instruction and its sentinel form a restartable sequence. When a sentinel reports an exception caused by a speculative instruction, re-execution beginning with the speculative instruction is possible. Re-execution proceeds by repairing the execution of the speculative instruction and using the reported program counter to branch back to the speculative instruction. All subsequent instructions will then be re-executed correctly.

[4] Note that the move instruction is necessary only if the incremented register is used before redefined after the program execution exits the current superblock.

4 Allowing Speculative Stores

A limitation of sentinel scheduling up to this point of discussion is that it does not allow speculative store instructions. In this section, an extension to sentinel scheduling is described which allows store instructions to move above branch instructions. In the following subsections, the additional architectural and compiler support required for speculative stores is presented.

4.1 Additional Architectural Support

In order to support speculatively executed store instructions, the operation of the data memory subsystem must be modified. In this discussion, it will be assumed that an N entry store buffer exists between the CPU and the data cache [14].

Operation of a Conventional Store Buffer. A store buffer has three primary functions. First, it creates a new entry for each store instruction executed by the CPU. Each store buffer entry consists of the store address, store data, and several status bits. Address translation is performed during insertion to determine if an exception (access violation or page fault) has occurred. If an exception occurs, it is handled immediately. The store buffer also supplies data to the CPU whenever a load with a matching address to a valid store buffer entry is executed. Finally, the store buffer releases entries to update the data cache. The store buffer operates as a first in first out circular queue. When the data cache is available and the buffer is not empty, the entry at the head of the queue is transferred to the data cache.

Operation of Store Buffer Supporting Speculative Stores. Speculative store instructions can be utilized if the store buffer is modified to allow probationary entries. Probationary entries are for speculative stores which may or may not require execution. Probationary entries are later confirmed by specific instructions if the predicted path of control is followed or invalidated when a branch direction is mispredicted. To support probationary entries, each store buffer entry requires three additional fields, a confirmation bit, an exception tag, and an exception pc. Also, an additional instruction to confirm store instructions in the store buffer, $confirm\_store(index)$, is needed. Finally, a mechanism to invalidate all probationary store buffer entries whenever a branch prediction miss occurs is required.

Each function of the store buffer requires some modifications to handle probationary entries. The insertion of a store into the store buffer is summarized in Table 2. Note that non-speculative stores enter the buffer as confirmed entries, while speculative stores enter as probationary entries. Also, when the buffer is full, the processor is stalled to wait for an entry to become available. When a load instruction is executed, both confirmed and unconfirmed entries are searched for a matching address. However, a probationary entry with its exception tag set will not participate in the search.[5] This exclusion from the search is to enable re-execution of the load instruction independent from re-execution of a matching excepting store in the store buffer. The releasing function of the store buffer is changed so that probationary stores are not allowed to update the data cache. This is accomplished by preventing any releases from the store buffer when the entry at the head of the buffer is probationary.

Two additional functions are required for the store buffer, confirming and cancelling probationary entries. A probationary store in the store buffer is confirmed by a $confirm\_store(index)$ instruction. The index signifies which entry is confirmed counting from the tail entry. If the exception tag of the entry being confirmed is set, an exception must be reported. The exception is handled in the same manner as when an exception occurs during insertion of a non-speculative store instruction. However, the pc of the excepting instruction is provided in the exception pc field of the particular store buffer entry. All probationary stores are cancelled when a mispredicted branch is detected. Cancellation of a probationary store is accomplished by resetting the valid bit of the corresponding store buffer entry.

4.2 Scheduling Support for Store Movement

An instruction scheduler can be extended to move store instructions above branch instructions in a straightforward manner. Stores are permitted to move above branches by removing control dependences between a store instruction and all preceding branch instructions during dependence graph reduction. Dependence reduction also marks all store instructions as unprotected.

The list scheduling phase must also be modified to insert special sentinel instructions for speculative stores. The $confirm\_store(index)$ instruction is inserted as the sentinel of a speculative store. Again, the sentinel is restricted to remain in the home block of the store. The value of $index$ is the number of stores (regular and speculative) between a speculative store and its corresponding confirm.

Exception detection is not impaired by the movement of stores. A store instruction will only be confirmed when the branches it moved across all have been predicted correctly at compile time. If any of the branches are incorrectly predicted, the store is cancelled. An exception for a speculative store is reported only at the time of confirmation, therefore only exceptions for those stores that are supposed to be executed will be reported. Also, the $confirm\_store$ instruction is restricted to remain in the home block of the store, thus exceptions occurring in different basic blocks will be reported in the proper order. Again, if multiple exceptions occur in the same basic block, the exceptions will be signaled, however they are not guaranteed in the order of the original code sequence.

A possible deadlock situation can occur when attempting to insert a store into the buffer if the store buffer is full and the entry at the head of the store buffer is unconfirmed. This situation can be prevented during scheduling by allowing a speculative store to be separated from its confirm by at most $N-1$ (for an N entry store buffer) stores. All probationary stores, therefore, must either be confirmed or cancelled within N stores of itself. The size of the store buffer, though, is now an architectural parameter that must be available to the scheduler.

[5] Note that an exception reflected in the exception tag of a probationary store buffer entry will be subsequently detected by the corresponding $confirm\_store$ instruction of the speculative store.

Function	Latency	Function	Latency
Int ALU	1	FP ALU	3
Int multiply	3	FP conversion	3
Int divide	10	FP multiply	3
branch	1 / 1 slot	FP divide	10
memory load	2	memory store	1

Table 3: Instruction latencies.

5 Experimental Evaluation

In this section, the effectiveness of sentinel scheduling is analyzed for a set of numeric and non-numeric benchmarks. The performance of sentinel scheduling and sentinel scheduling with speculative stores is compared with restricted and general percolation.

5.1 Methodology

Sentinel superblock scheduling has been implemented in the instruction scheduler of IMPACT-I compiler. The IMPACT-I compiler is a prototype optimizing compiler designed to generate efficient code for VLIW and superscalar processors [8]. A superblock is the basic scope for the instruction scheduler.

The instruction scheduler takes as an input a machine description file that characterizes the instruction set, the microarchitecture (including the number of instructions that can be fetched/issued in a cycle and the instruction latencies), and the code scheduling model. The underlying microarchitecture is assumed to have CRAY-1 style interlocking and deterministic instruction latencies (Table 3). The instruction set is a RISC assembly language similar to the MIPS R2000 instruction set. The basic processor has 64 integer registers, 64 floating point registers, and an 8 entry store buffer. The basic processor is assumed to trap on exceptions for memory load, memory store, integer divide, and all floating point instructions.

For each machine configuration, the program execution time, assuming a 100% cache hit rate, is derived from execution-driven simulation. The benchmarks used in this study consist of 5 numeric and 12 non-numeric programs. The numeric programs are all from the SPEC suite, *doduc*, *fpppp*, *matrix300*, *nasa7*, and *tomcatv*. The non-numeric programs consist of 3 programs from the SPEC suite, *eqntott*, *espresso*, and *xlisp*; and 9 other commonly used non-numeric programs, *cccp*, *cmp*, *compress*, *eqn*, *grep*, *lex*, *tbl*, *wc*, and *yacc*.

spec	*src(I).except_tag* †	*I causes except* ‡	*description*
0	0	0	insert a non-speculative store as a confirmed entry
0	0	1	force all confirmed entries at head of buffer to update cache, save contents of store buffer ◇, process exception
0	1	0	signal exception, report pc = $src(I).data$
0	1	1	signal exception, report pc = $src(I).data$
1	0	0	insert speculative store as a pending entry
1	0	1	insert speculative store as a pending entry, set exception tag, set exception pc to pc of I
1	1	0	insert speculative store as a pending entry, set exception tag, set exception pc to $src(I).data$
1	1	1	insert speculative store as a pending entry, set exception tag, set exception pc to $src(I).data$

† Instruction producing source operand of store contains exception condition, so store must just propagate the exception.
‡ The store instruction results in an exception.
◇ Saving the contents of the store buffer only necessary when speculative stores are allowed.

Table 2: Insertion of store into store buffer.

5.2 Results

In this section the performance of the varying scheduling models is compared for VLIW/superscalar processors with issue rates 2, 4, and 8. The issue rate is the maximum number of instructions the processor can fetch and issue per cycle. No limitation has been placed on the combination of instructions that can be issued in the same cycle. The base machine for all speedup calculations has an issue rate of 1 and supports the restricted percolation scheduling model for speculative code motion. Note that the experiments do not take into account compiler constraints to ensure recovery. These constraints are expected to reduce the performance of the sentinel scheduling model. We are currently quantifying this performance impact.

Comparison of Sentinel Scheduling and Restricted Percolation. The performance of the sentinel scheduling model and the restricted percolation scheduling model is compared in Figure 4. For the non-numeric programs, sentinel scheduling provides large performance improvements over restricted percolation. For example, an issue 8 processor with sentinel scheduling support achieves from 18% to 135% speedup improvement, with an average of 57%, over restricted percolation. The ability to speculatively execute potential exception-causing instructions allows the scheduler to exploit higher levels of ILP. Without sentinel scheduling support, the scheduler is most restricted by not being able to schedule load instructions speculatively. Load instructions are often the first instruction in a long chain of dependent instructions. Thus, the ability to speculatively schedule load instructions as early as possible is extremely important for VLIW and superscalar processors. The importance of sentinel scheduling support also grows for higher issue rate processors.

Numeric programs with large numbers of conditional branches can be expected to achieve similar performance gains to non-numeric programs with sentinel scheduling support, whereas numeric programs with few conditional branches will likely achieve only moderate performance improvements with this support. For *fpppp*, *matrix*300, and *nasa*7, few conditional branches are present in the most important program sections, thus the need for speculative support is not as important. For these programs restricted percolation already achieves a high instruction execution rate (Figure 4). However, for *doduc* and *tomcatv* which contain conditional branches in frequently executed program sections, sentinel scheduling provides significant performance for all issue rates. For example with an issue 4 processor, sentinel scheduling support increases performance by 36% and 38% for *doduc* and *tomcatv*, respectively (Figure 4).

Comparison of Sentinel Scheduling and General Percolation. The performance of the sentinel scheduling model and the general percolation scheduling model is compared in Figure 5. From the figure, it can be seen that the performance of sentinel scheduling is almost identical to the performance of general percolation for all issue rates. Sentinel scheduling requires additional sentinel instructions to be inserted when unprotected instructions are speculatively scheduled. However, most of the sentinels can be scheduled in empty instruction slots so they do not cause significant execution overhead. The largest performance difference occurs for *grep* for an issue 2 processor due to a lack of available slots to insert the sentinels. For an issue 8 processor, though, no performance loss is observed.

Effectiveness of Sentinel Scheduling with Speculative Stores. The performance of sentinel scheduling with speculative store support is also shown in Figure 5. For an issue 8 processor, an average of 7.4% improvement for non-numeric programs and 2.6% for numeric programs is observed. For the non-numeric programs, *cmp* and *grep* achieve over 20% performance gain for an issue 4 and issue 8 processor. Moderate performance gains are observed for *cccp*, *compress*, *eqn*, *espresso*, *lex*, *tbl*, *xlisp*, and *yacc*. No performance improvement is obtained for *eqntott* and *wc*. This lack of improvement is due to few store instructions in the most frequently executed code sequences with conditional branches. For the numeric programs, moderate performance gains are observed for *doduc* and *nasa*7, while no performance improvement is observed for *fpppp*, *matrix*300,

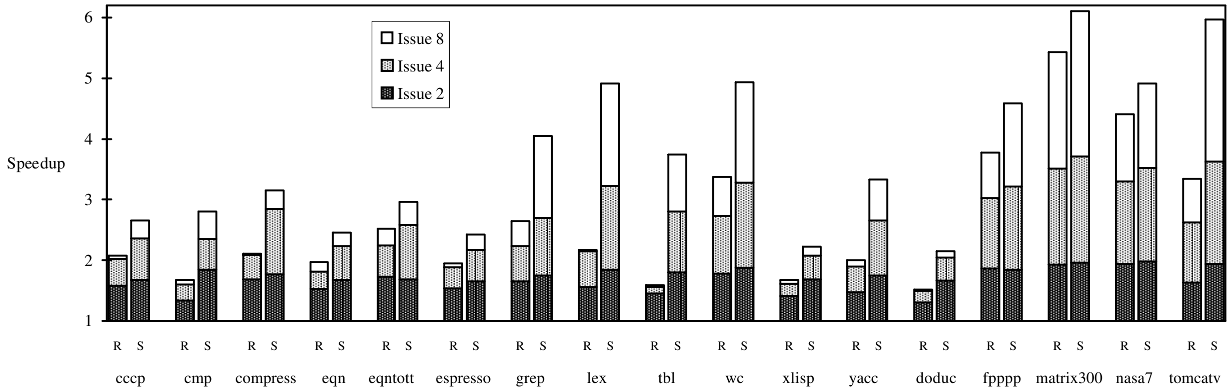

Figure 4: Performance comparison of sentinel scheduling (S) and restricted percolation (R).

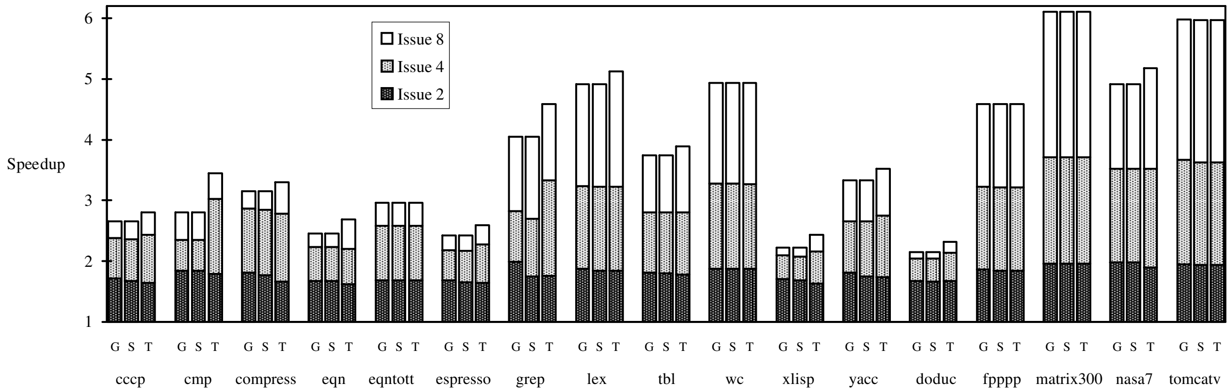

Figure 5: Performance comparison of sentinel scheduling (S), sentinel scheduling with speculative stores (T), and general percolation (G).

and *tomcatv*. Similarly, this lack of improvement is due to few store instructions in frequently executed code sequences with conditional branches.

6 Conclusions

In this paper, a set of architectural and compiler support, referred to as sentinel scheduling, is introduced. Sentinel scheduling provides an effective framework for compiler-controlled speculative execution that accurately detects and reports all exceptions. Whenever a potential exception-causing instruction is speculatively executed, the scheduler ensures that a non-speculative sentinel instruction remains in the home block of the instruction to check if an exception occurred. Sentinel scheduling is shown to provide substantial performance improvements for both non-numeric and numeric programs. For an issue 8 processor, an average performance improvement of 57% for non-numeric programs and 32% for numeric programs is achieved over the restricted code percolation scheduling model. Also, the performance of sentinel scheduling is shown to be almost identical to that of general percolation. This confirms that the constraint of properly reporting all exceptions does not restrict the performance of sentinel scheduling.

An extension of sentinel scheduling to allow speculative execution of store instructions is also described. A store buffer which allows probationary entries is proposed. Speculative store instructions enter the store buffer as probationary entries and are later confirmed by explicit instructions inserted by the scheduler. With speculative store support, an average performance improvement of 7.4% for non-numeric programs and 2.6% for numeric programs is observed. The performance improvement, however, varies the frequency of stores and conditional branches in the benchmarks.

Acknowledgements

The authors would like to thank Thomas Conte, Sadun Anik, Richard Hank, and Roger Bringmann, along with all members of the IMPACT research group for their comments and suggestions. Special thanks Pohua Chang at Intel Corporation, David Callahan at Tera Computer Corporation, and to the anonymous referees whose comments and suggestions helped to improve the quality of this paper significantly. This research has been supported by JSEP under Contract N00014-90-J-1270, Dr. Lee Hoevel at NCR, the AMD 29K Advanced Processor Development Division, Matsushita Electric Industrial Co. Ltd., Hewlett-Packard, and NASA under Contract NASA NAG 1-613 in cooperation with ICLASS.

References

[1] 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.

[2] 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.

[3] 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.

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

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

[6] 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.

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

[8] 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.

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

[10] K. Ebcioglu, "A compilation technique for software pipelining of loops with conditional jumps," in *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture*, pp. 69–79, December 1987.

[11] 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.

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

[13] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Exception recovery for systems with compiler-controlled speculative execution," tech. rep., Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, in preparation 1992.

[14] W. M. Johnson, *Superscalar Microprocessor Design.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1991.

Appendix

An algorithm to perform sentinel superblock scheduling is presented below. The *dependence_graph_reduction* function removes control dependences from a dependence graph to allow speculative code motion. Instructions may be speculatively executed if they are allowed as speculative in the architecture. For example branches, subroutine calls, and i/o instructions may not be speculatively executed. Also store instructions without support for probationary stores are not allowed to be speculative. The *sentinel_scheduling* function utilizes the dependence graph to perform list scheduling of all instructions in a superblock.

```
dependence_graph_reduction(superblock) {
  /* Remove control deps to allow upward code motion */
  for each instruction in sequential order in superblock, I {
    if (I is unprotected) {
      if (dest(I) is used at or before the first succeeding
          control instruction) {
        mark I as protected
        mark use as unprotected }
      if (I allowed to be speculative) {
        for each branch before I, BR {
          if (dest(I) not live when BR is taken) {
            remove control dependence from BR to I } } } }
    else if (I may cause an exception) {
      if (dest(I) is used at or before the first succeeding
          control instruction) {
        mark use as unprotected }
      else {
        mark I as unprotected }
      if (I allowed to be speculative) {
        for each branch before I, BR {
          if (dest(I) not live when BR is taken) {
            remove control dependence from BR to I } } } }
    else if (I allowed to be speculative) {
      for each branch before I, BR {
        if (dest(I) not live when BR is taken) {
          remove control dependence from BR to I } } } } }

sentinel_scheduling(superblock) {
  construct dependence graph
  mark all instructions in superblock as protected
  dependence_graph_reduction(superblock)
  /* List schedule all instructions in the superblock */
  for each instruction in superblock, I {
    compute priority of I
    add I to set of unscheduled instructions }
  while (unscheduled set of instruction is not empty) {
    active_set = set of unscheduled instructions that are ready
    sort active_set according to instruction priorities
    find the best set of instructions to issue from active_set
    for each instruction issued at the current cycle, I {
      if (I is moved above a branch) {
        set speculative modifier of I }
      if ((I is speculative) and (I is unprotected)) {
        insert a check_exception(dest(I)) instruction, J,
            into superblock
        add a flow dependence from I to J
        add a control dependence from the first branch I
            moved above to J
        add a control dependence from J to the first branch
            originally below I
        add J to unscheduled set of instructions }
      remove I from set of unscheduled instructions } } }
```